VISIT…



BRIEF VERSION



INTRODUCTION TO JAVA™ PROGRAMMING

EIGHTH EDITION



Y. Daniel Liang

INTRODUCTION TO JAVA™ PROGRAMMING

This page intentionally left blank

INTRODUCTION TO JAVA™ PROGRAMMING

BRIEF VERSION

Eighth Edition

Y. Daniel Liang

Armstrong Atlantic State University

Prentice Hall

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Vice President and Editorial Director, ECS: Marcia J. Horton
Editor in Chief, Computer Science: Michael Hirsch
Executive Editor: Tracy Dunkelberger
Assistant Editor: Melinda Haggerty
Editorial Assistant: Allison Michael
Vice President, Production: Vince O'Brien
Senior Managing Editor: Scott Disanno
Production Editor: Irwin Zucker
Senior Operations Specialist: Alan Fischer
Marketing Manager: Erin Davis
Marketing Assistant: Mack Patterson
Art Director: Kenny Beck
Cover Image: Male Ruby-throated Hummingbird / Steve Byland / Shutterstock;
Hummingbird, Nazca Lines / Gary Yim / Shutterstock
Art Editor: Greg Dulles
Media Editor: Daniel Sandin

Copyright © 2011, 2009, 2007, 2004 by Pearson Higher Education. Upper Saddle River, New Jersey, 07458.
All right reserved. Manufactured in the United States of America. This publication is protected by Copyright and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission(s) to use materials from this work, please submit a written request to Pearson Higher Education, Permissions Department, 1 Lake Street, Upper Saddle River, NJ 07458.

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Library of Congress Cataloging-in-Publication Data on file.

Prentice Hall
is an imprint of

PEARSON

www.pearsonhighered.com

10 9 8 7 6 5 4 3 2 1

ISBN-13: 978-0-13-213079-0
ISBN-10: 0-13-213079-3

This book is dedicated to Dr. S. K. Dhall and Dr. S. Lakshmivarahan of the University of Oklahoma, who inspired me in teaching and research. Thank you for being my mentors and advisors.

To Samantha, Michael, and Michelle

This page intentionally left blank

PREFACE

This book is a brief version of *Introduction to Java Programming, Comprehensive Version, 8E*. This version is designed for an introductory programming course, commonly known as *CS1*. This version contains the first twenty chapters in the comprehensive version. brief version comprehensive version

This book uses the fundamentals-first approach and teaches programming concepts and techniques in a problem-driven way.

The fundamentals-first approach introduces basic programming concepts and techniques before objects and classes. My own experience, confirmed by the experiences of many colleagues, demonstrates that new programmers in order to succeed must learn basic logic and fundamental programming techniques such as loops and stepwise refinement. The fundamental concepts and techniques of loops, methods, and arrays are the foundation for programming. Building the foundation prepares students to learn object-oriented programming, GUI, database, and Web programming. fundamentals-first

Problem-driven means focused on problem-solving rather than syntax. We make introductory programming interesting by using interesting problems. The central thread of this book is on solving problems. Appropriate syntax and library are introduced to support the writing of a program for solving the problems. To support teaching programming in a problem-driven way, the book provides a wide variety of problems at various levels of difficulty to motivate students. In order to appeal to students in all majors, the problems cover many application areas in math, science, business, financials, gaming, animation, and multimedia. problem-driven

What's New in This Edition?

This edition substantially improves *Introduction to Java Programming, Seventh Edition*. The major improvements are as follows:

- This edition is completely revised in every detail to enhance clarity, presentation, content, examples, and exercises. complete revision
- In the examples and exercises, which are provided to motivate and stimulate student interest in programming, one-fifth of the problems are new. new problems
- In the previous edition, console input was covered at the end of Chapter 2. The new edition introduces console input early in Chapter 2 so that students can write interactive programs early. early console input
- The hand trace box is added for many programs in early chapters to help noive students to read and trace programs. hand trace box
- Single-dimensional arrays and multidimensional arrays are covered in two chapters to give instructors the flexibility to cover multidimensional arrays later. multidimensional arrays
- The case study for the Sudoku problem has been moved to the Companion Website. A more pedagogically effective simple version of the Sudoku problem is presented instead. Sudoku problem simplified
- The design of the API for Java GUI programming is an excellent example of how the object-oriented principle is applied. Students learn better with concrete and visual examples. So basic GUI now precedes the introduction of abstract classes and interfaces. The instructor, however, can still choose to cover abstract classes and interfaces before GUI. basic GUI earlier

exception handling earlier

- Exception handling is covered before abstract classes and interfaces. The instructor can still choose to cover exception handling later.

design guidelines

- Chapter 12, "Object-Oriented Design and Patterns," in the previous edition has been replaced by spreading the design guidelines and patterns into several chapters so that these topics can be covered in appropriate context.

Learning Strategies

learn from mistakes

A programming course is quite different from other courses. In a programming course, you learn from examples, from practice, and *from mistakes*. You need to devote a lot of time to writing programs, testing them, and fixing errors.

programmatic solution

For first-time programmers, learning Java is like learning any high-level programming language. The fundamental point is to develop the critical skills of formulating programmatic solutions for real problems and translating them into programs using selection statements, loops, methods, and arrays.

object-oriented programming

Once you acquire the basic skills of writing programs using loops, methods, and arrays, you can begin to learn how to develop large programs and GUI programs using the object-oriented approach.

Java API

When you know how to program and you understand the concept of object-oriented programming, learning Java becomes a matter of learning the Java API. The Java API establishes a framework for programmers to develop applications using Java. You have to use the classes and interfaces in the API and follow their conventions and rules to create applications. The best way to learn the Java API is to imitate examples and do exercises.

Pedagogical Features

The book uses the following elements to get the most from the material:

- **Objectives** list what students should have learned from the chapter. This will help them determine whether they have met the objectives after completing the chapter.
- **Introduction** opens the discussion with representative problems to give the reader an overview of what to expect from the chapter.
- **Problems** carefully chosen and presented in an easy-to-follow style, teach problem solving and programming concepts. The book uses many small, simple, and stimulating examples to demonstrate important ideas.
- **Chapter Summary** reviews the important subjects that students should understand and remember. It helps them reinforce the key concepts they have learned in the chapter.
- **Review Questions** are grouped by sections to help students track their progress and evaluate their learning.
- **Programming Exercises** are grouped by sections to provide students with opportunities to apply on their own the new skills they have learned. The level of difficulty is rated as easy (no asterisk), moderate (*), hard (**), or challenging (***). The trick of learning programming is practice, practice, and practice. To that end, the book provides a great many exercises.
- **LiveLab** is a programming course assessment and management system. Students can submit programs/quizzes online. The system automatically grades the programs/quizzes and gives students instant feedback.
- **Notes**, **Tips**, and **Cautions** are inserted throughout the text to offer valuable advice and insight on important aspects of program development.



Note
Provides additional information on the subject and reinforces important concepts.



Tip
Teaches good programming style and practice.



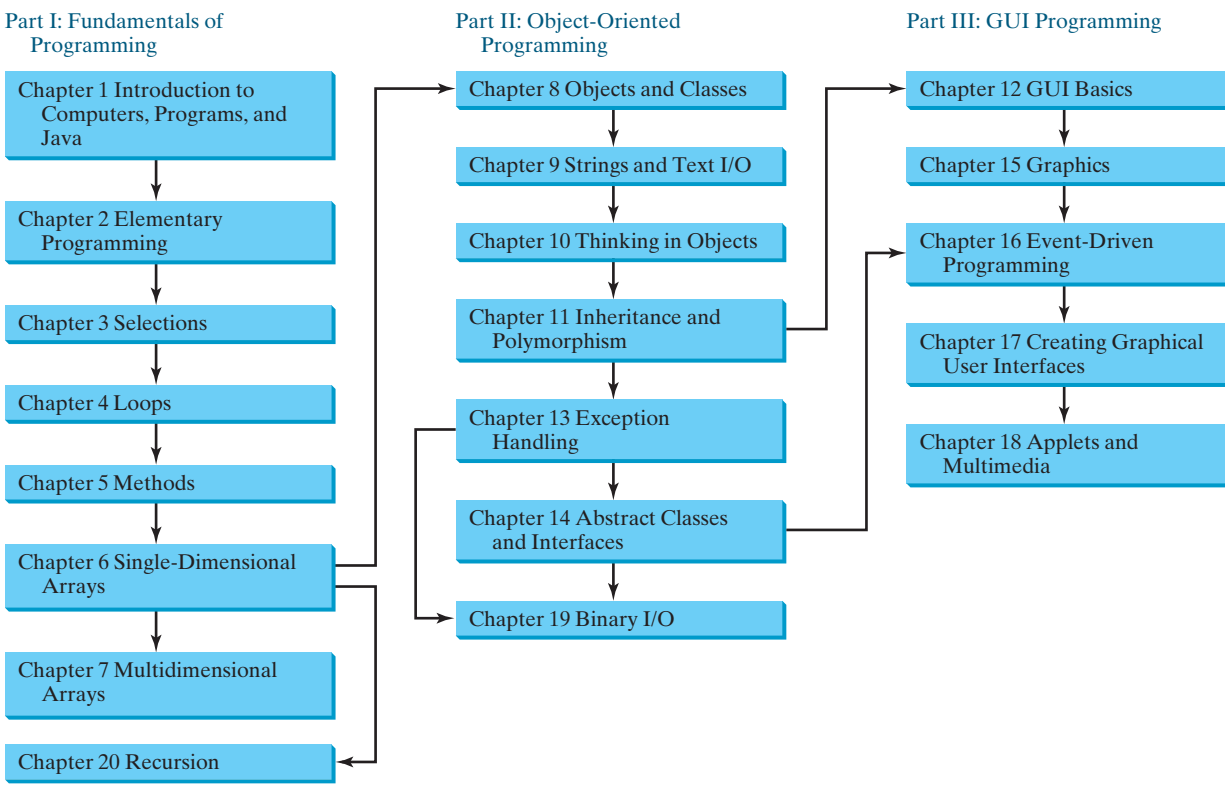
Caution
Helps students steer away from the pitfalls of programming errors.



Design Guide
Provides the guidelines for designing programs.

Flexible Chapter Orderings

The book is designed to provide flexible chapter orderings to enable GUI, exception handling, and recursion to be covered earlier or later. The diagram shows the chapter dependencies.


Part I: Fundamentals of Programming
Chapter 1 Introduction to Computers, Programs, and Java
Chapter 2 Elementary Programming
Chapter 3 Selections
Chapter 4 Loops
Chapter 5 Methods
Chapter 6 Single-Dimensional Arrays
Chapter 7 Multidimensional Arrays
Chapter 20 Recursion
Part II: Object-Oriented Programming
Chapter 8 Objects and Classes
Chapter 9 Strings and Text I/O
Chapter 10 Thinking in Objects
Chapter 11 Inheritance and Polymorphism
Chapter 13 Exception Handling
Chapter 14 Abstract Classes and Interfaces
Chapter 19 Binary I/O
Part III: GUI Programming
Chapter 12 GUI Basics
Chapter 15 Graphics
Chapter 16 Event-Driven Programming
Chapter 17 Creating Graphical User Interfaces
Chapter 18 Applets and Multimedia

Organization of the Book

The chapters in the brief version can be grouped into three parts that, taken together, form a solid introduction to Java programming. Because knowledge is cumulative, the early chapters provide the conceptual basis for understanding programming and guide students through

simple examples and exercises; subsequent chapters progressively present Java programming in detail, culminating with the development of comprehensive Java applications.

Part I: Fundamentals of Programming (Chapters 1–7, 20)

The first part of the book is a stepping stone, preparing you to embark on the journey of learning Java. You will begin to know Java (Chapter 1), and will learn fundamental programming techniques with primitive data types, variables, constants, expressions, and operators (Chapter 2), control statements (Chapters 3–4), methods (Chapter 5), and arrays (Chapters 6–7). After Chapter 6, you may jump to Chapter 20 to learn how to write recursive methods for solving inherently recursive problems.

Part II: Object-Oriented Programming (Chapters 8–11, 13–14, 19)

This part introduces object-oriented programming. Java is an object-oriented programming language that uses abstraction, encapsulation, inheritance, and polymorphism to provide great flexibility, modularity, and reusability in developing software. You will learn programming with objects and classes (Chapters 8–10), class inheritance (Chapter 11), polymorphism (Chapter 11), exception handling (Chapter 13), abstract classes (Chapter 14), and interfaces (Chapter 14). Processing strings will be introduced in Chapter 9 along with text I/O. Binary I/O is introduced in Chapter 19.

Part III: GUI Programming (Chapters 12, 15–18)

This part introduces elementary Java GUI programming in Chapters 12 and 15–18. Major topics include GUI basics (Chapter 12), drawing shapes (Chapter 15), event-driven programming (Chapter 16), creating graphical user interfaces (Chapter 17), and writing applets (Chapter 18). You will learn the architecture of Java GUI programming and use the GUI components to develop applications and applets from these elementary GUI chapters.

Java Development Tools

IDE tutorials

You can use a text editor, such as the Windows Notepad or WordPad, to create Java programs and to compile and run the programs from the command window. You can also use a Java development tool, such as TextPad, NetBeans, or Eclipse. These tools support an integrated development environment (IDE) for rapidly developing Java programs. Editing, compiling, building, executing, and debugging programs are integrated in one graphical user interface. Using these tools effectively can greatly increase your programming productivity. TextPad is a primitive IDE tool. NetBeans and Eclipse are more sophisticated, but they are easy to use if you follow the tutorials. Tutorials on TextPad, NetBeans, and Eclipse can be found in the supplements on the Companion Website.

LiveLab

This book is accompanied by an improved faster Web-based course assessment and management system. The system has three main components:

- **Automatic Grading System:** It can automatically grade exercises from the text or created by instructors.
- **Quiz Creation/Submission/Grading System:** It enables instructors to create/modify quizzes that students can take and be graded upon automatically.
- **Tracking grades, attendance, etc:** The system enables the students to track grades and instructors to view the grades of all students, and to track attendance.

The main features of the Automatic Grading System are as follows:

- Allows students to compile, run and submit exercises. (The system checks whether their program runs correctly—students can continue to run and submit the program before the due date.)
- Allows instructors to review submissions; run programs with instructor test cases; correct them; and provide feedback to students.
- Allows instructors to create/modify their own exercises, create public and secret test cases, assign exercises, and set due dates for the whole class or for individuals.
- All the exercises in the text can be assigned to students. Additionally, LiveLab provides extra exercises that are not printed in the text.
- Allows instructors to sort and filter all exercises and check grades (by time frame, student, and/or exercise).
- Allows instructors to delete students from the system.
- Allows students and instructors to track grades on exercises.

The main features of the Quiz System are as follows:

- Allows instructors to create/modify quizzes from test bank or a text file or to create complete new tests online.
- Allows instructors to assign the quizzes to students and set a due date and test time limit for the whole class or for individuals.
- Allows students and instructors to review submitted quizzes.
- Allows students and instructors to track grades on quizzes.

Video Notes are Pearson's new visual tool designed for teaching students key programming concepts and techniques. These short step-by-step videos demonstrate how to solve problems from design through coding. Video Notes allows for self-paced instruction with easy navigation including the ability to select, play, rewind, fast-forward, and stop within each Video Note exercise.

Video Note margin icons in your textbook let you know what a Video Notes video is available for a particular concept or homework problem.

Video Notes are free with the purchase of a new textbook. To purchase access to Video Notes, please go to www.pearsonhighered.com/liang.

Student Resource Materials

The student resources can be accessed through the Publisher's Web site (www.pearsonhighered.com/liang) and the Companion Web site (www.cs.armstrong.edu/liang/intro8e). The resources include:

- Answers to review questions
- Solutions to even-numbered programming exercises
- Source code for book examples
- Interactive self-test (organized by chapter sections)
- LiveLab
- Resource links
- Errata

- Video Notes
- Web Chapters

To access the Video Notes and Web Chapters, students must log onto www.pearsonhighered.com/liang and use the access card located in the front of the book to register and access the material. If there is no access card in the front of this textbook, students can purchase access by visiting www.pearsonhighered.com/liang and selecting *purchase access to premium content*.

Additional Supplements

The text covers the essential subjects. The supplements extend the text to introduce additional topics that might be of interest to readers. The supplements listed in this table are available from the Companion Web site.

Supplements on the Companion Web site

Part I General Supplements
- A Glossary
- B Installing and Configuring JDK
- C Compiling and Running Java from the Command Window
- D Java Coding Style Guidelines
- E Creating Desktop Shortcuts for Java Applications on Windows
- F Using Packages to Organize the Classes in the Text

Part II IDE Supplements
- A TextPad Tutorial
- B NetBeans Tutorial | One Page Startup Instruction
- C Learning Java Effectively with NetBeans
- D Eclipse Tutorial | One Page Startup Instruction
- E Learning Java Effectively with Eclipse

Part III Java Supplements
- A Java Characteristics
- B Discussion on Operator and Operand Evaluations
- C The & and | Operators
- D Bitwise Operations
- E Statement Labels with break and continue
- F Enumerated Types
- G Packages
- H Regular Expressions
- I Formatted Strings
- J The Methods in the Object Class
- K Hiding Data Fields and Static Methods
- L Initialization Blocks
- M Extended Discussions on Overriding Methods
- N Design Patterns
- O Text I/O Prior to JDK 1.5 (Reader and Writer Classes)
- P Assertions
- Q Packaging and Deploying Java Projects
- R Java Web Start
- S GridBagLayout | OverlayLayout | SpringLayout
- T Networking Using Datagram Protocol
- U Creating Internal Frames
- V Pluggable Look and Feel
- W UML Graphical Notations
- X Testing Classes Using JUnit
- Y JNI
- Z The StringTokenizer Class

Part IV Database Supplements
- A SQL Statements for Creating and Initializing Tables Used in the Book
- B MySQL Tutorial
- C Oracle Tutorial
- D Microsoft Access Tutorial
- E Introduction to Database Systems
- F Relational Database Concept
- G Database Design
- H SQL Basics
- I Advanced SQL

Part V Web Programming Supplements
- A HTML and XHTML Tutorial
- B CSS Tutorial
- C XML
- D Java and XML
- E Tomcat Tutorial
- F More Examples on JSF and Visual Web Development

Instructor Resource Materials

The instructor resources can be accessed through the Publisher's Web site (www.pearsonhighered.com/liang) and the Companion Web site (www.cs.armstrong.edu/liang/intro8e). For username and password information to the Liang 8e site, please contact your Pearson Representative.

The resources include:

- PowerPoint lecture slides with source code and run program capacity
- Instructor solutions manual
- Computerized test generator
- Sample exams using multiple choice and short answer questions, write and trace programs, and correcting programming errors.
- LiveLab
- Errata
- Video Notes
- Web Chapters

To access the Video Notes and Web Chapters, instructors must log onto www.pearsonhighered.com/liang and register.

Acknowledgments

I would like to thank Armstrong Atlantic State University for enabling me to teach what I write and for supporting me in writing what I teach. Teaching is the source of inspiration for continuing to improve the book. I am grateful to the instructors and students who have offered comments, suggestions, bug reports, and praise.

This book has been greatly enhanced thanks to outstanding reviews for this and previous editions. The reviewers are: Elizabeth Adams (James Madison University), Syed Ahmed (North Georgia College and State University), Omar Aldawud (Illinois Institute of Technology), Yang Ang (University of Wollongong, Australia), Kevin Bierre (Rochester Institute of Technology), David Champion (DeVry Institute), James Chegwidden (Tarrant County College), Anup Dargar (University of North Dakota), Charles Dierbach (Towson University), Frank Ducrest (University of Louisiana at Lafayette), Erica Eddy (University of Wisconsin at Parkside), Deena Engel (New York University), Henry A Etlinger (Rochester Institute of Technology), James Ten Eyck (Marist College), Olac Fuentes (University of Texas at El Paso), Harold Grossman (Clemson University), Barbara Guillot (Louisiana State University), Ron Hofman (Red River College, Canada), Stephen Hughes (Roanoke College), Vladan Jovanovic (Georgia Southern University), Edwin Kay (Lehigh University), Larry King (University of Texas at Dallas), Nana Kofi (Langara College, Canada), George Koutsogiannakis (Illinois Institute of Technology), Roger Kraft (Purdue University at Calumet), Hong Lin (DeVry Institute), Dan Lipsa (Armstrong Atlantic State University), James Madison (Rensselaer Polytechnic Institute), Frank Malinowski (Darton College), Tim Margush (University of Akron), Debbie Masada (Sun Microsystems), Blayne Mayfield (Oklahoma State University), John McGrath (J.P. McGrath Consulting), Shyamal Mitra (University of Texas at Austin), Michel Mitri (James Madison University), Kenrick Mock (University of Alaska Anchorage), Jun Ni (University of Iowa), Benjamin Nystuen (University of Colorado at Colorado Springs), Maureen Opkins (CA State University, Long Beach), Gavin Osborne (University of Saskatchewan), Kevin Parker (Idaho State University), Dale Parson (Kutztown University), Mark Pendergast (Florida Gulf Coast

University), Richard Povinelli (Marquette University), Roger Priebe (University of Texas at Austin), Mary Ann Pumphrey (De Anza Junior College), Pat Roth (Southern Polytechnic State University), Ronald F. Taylor (Wright State University), Carolyn Schauble (Colorado State University), David Scuse (University of Manitoba), Ashraf Shirani (San Jose State University), Daniel Spiegel (Kutztown University), Amr Sabry (Indiana University), Lixin Tao (Pace University), Russ Tront (Simon Fraser University), Deborah Trytten (University of Oklahoma), Kent Vidrine (George Washington University), and Bahram Zartoshty (California State University at Northridge).

It is a great pleasure, honor, and privilege to work with Pearson. I would like to thank Tracy Dunkelberger and her colleagues Marcia Horton, Margaret Waples, Erin Davis, Michael Hirsh, Matt Goldstein, Jake Warde, Melinda Haggerty, Allison Michael, Scott Disanno, Irwin Zucker, and their colleagues for organizing, producing, and promoting this project, and Robert Lentz for copy editing.

As always, I am indebted to my wife, Samantha, for her love, support, and encouragement.

Y. Daniel Liang
y.daniel.liang@gmail.com
www.cs.armstrong.edu/liang
www.pearsonhighered.com/liang

BRIEF CONTENTS

1	Introduction to Computers, Programs, and Java	1
2	Elementary Programming	23
3	Selections	71
4	Loops	115
5	Methods	155
6	Single-Dimensional Arrays	197
7	Multidimensional Arrays	235
8	Objects and Classes	263
9	Strings and Text I/O	301
10	Thinking in Objects	343
11	Inheritance and Polymorphism	373
12	GUI Basics	405
13	Exception Handling	431
14	Abstract Classes and Interfaces	457
15	Graphics	497
16	Event-Driven Programming	533
17	Creating Graphical User Interfaces	571
18	Applets and Multimedia	613
19	Binary I/O	649
20	Recursion	677

APPENDIXES

A	Java Keywords	707
B	The ASCII Character Set	710
C	Operator Precedence Chart	712
D	Java Modifiers	714
E	Special Floating-Point Values	716
F	Number Systems	717

INDEX 721

CONTENTS

Chapter 1 Introduction to Computers, Programs, and Java 1

1.1 Introduction 2
1.2 What Is a Computer? 2
1.3 Programs 5
1.4 Operating Systems 7
1.5 Java, World Wide Web, and Beyond 8
1.6 The Java Language Specification, API, JDK, and IDE 10
1.7 A Simple Java Program 11
1.8 Creating, Compiling, and Executing a Java Program 13
1.9 (GUI) Displaying Text in a Message Dialog Box 16

Chapter 2 Elementary Programming 23

2.1 Introduction 24
2.2 Writing Simple Programs 24
2.3 Reading Input from the Console 26
2.4 Identifiers 29
2.5 Variables 29
2.6 Assignment Statements and Assignment Expressions 30
2.7 Named Constants 31
2.8 Numeric Data Types and Operations 32
2.9 Problem: Displaying the Current Time 37
2.10 Shorthand Operators 39
2.11 Numeric Type Conversions 41
2.12 Problem: Computing Loan Payments 43
2.13 Character Data Type and Operations 44
2.14 Problem: Counting Monetary Units 47
2.15 The `String` Type 50
2.16 Programming Style and Documentation 51
2.17 Programming Errors 53
2.18 (GUI) Getting Input from Input Dialogs 55

Chapter 3 Selections 71

3.1 Introduction 72
3.2 `boolean` Data Type 72
3.3 Problem: A Simple Math Learning Tool 73
3.4 `if` Statements 74

3.5 Problem: Guessing Birthdays 75
3.6 Two-Way `if` Statements 79
3.7 Nested `if` Statements 80
3.8 Common Errors in Selection Statements 81
3.9 Problem: An Improved Math Learning Tool 82
3.10 Problem: Computing Body Mass Index 84
3.11 Problem: Computing Taxes 85
3.12 Logical Operators 88
3.13 Problem: Determining Leap Year 90
3.14 Problem: Lottery 91
3.15 `switch` Statements 93
3.16 Conditional Expressions 95
3.17 Formatting Console Output 95
3.18 Operator Precedence and Associativity 97
3.19 (GUI) Confirmation Dialogs 98

Chapter 4 Loops 115
4.1 Introduction 116
4.2 The `while` Loop 116
4.3 The `do-while` Loop 124
4.4 The `for` Loop 126
4.5 Which Loop to Use? 128
4.6 Nested Loops 129
4.7 Minimizing Numeric Errors 130
4.8 Case Studies 131
4.9 Keywords `break` and `continue` 135
4.10 (GUI) Controlling a Loop with a Confirmation Dialog 139

Chapter 5 Methods 155
5.1 Introduction 156
5.2 Defining a Method 156
5.3 Calling a Method 158
5.4 `void` Method Example 160
5.5 Passing Parameters by Values 162
5.6 Modularizing Code 165
5.7 Problem: Converting Decimals to Hexadecimals 167
5.8 Overloading Methods 168
5.9 The Scope of Variables 171
5.10 The `Math` Class 172
5.11 Case Study: Generating Random Characters 175
5.12 Method Abstraction and Stepwise Refinement 176

Chapter 6 Single-Dimensional Arrays 197

6.1 Introduction 198
6.2 Array Basics 198
6.3 Problem: Lotto Numbers 204
6.4 Problem: Deck of Cards 206
6.5 Copying Arrays 208
6.6 Passing Arrays to Methods 209
6.7 Returning an Array from a Method 212
6.8 Variable-Length Argument Lists 215
6.9 Searching Arrays 216
6.10 Sorting Arrays 219
6.11 The `Arrays` Class 223

Chapter 7 Multidimensional Arrays 235

7.1 Introduction 236
7.2 Two-Dimensional Array Basics 236
7.3 Processing Two-Dimensional Arrays 238
7.4 Passing Two-Dimensional Arrays to Methods 240
7.5 Problem: Grading a Multiple-Choice Test 241
7.6 Problem: Finding a Closest Pair 242
7.7 Problem: Sudoku 244
7.8 Multidimensional Arrays 248

Chapter 8 Objects and Classes 263

8.1 Introduction 264
8.2 Defining Classes for Objects 264
8.3 Example: Defining Classes and Creating Objects 266
8.4 Constructing Objects Using Constructors 270
8.5 Accessing Objects via Reference Variables 270
8.6 Using Classes from the Java Library 274
8.7 Static Variables, Constants, and Methods 278
8.8 Visibility Modifiers 282
8.9 Data Field Encapsulation 283
8.10 Passing Objects to Methods 286
8.11 Array of Objects 287

Chapter 9 Strings and Text I/O 301

9.1 Introduction 302
9.2 The `String` Class 302
9.3 The `Character` Class 313

9.4 The StringBuilder/StringBuffer Class 315
9.5 Command-Line Arguments 320
9.6 The File Class 322
9.7 File Input and Output 325
9.8 (GUI) File Dialogs 329

Chapter 10 Thinking in Objects 343
10.1 Introduction 344
10.2 Immutable Objects and Classes 344
10.3 The Scope of Variables 345
10.4 The this Reference 346
10.5 Class Abstraction and Encapsulation 347
10.6 Object-Oriented Thinking 351
10.7 Object Composition 353
10.8 Designing the Course Class 355
10.9 Designing a Class for Stacks 357
10.10 Designing the GuessDate Class 359
10.11 Class Design Guidelines 362

Chapter 11 Inheritance and Polymorphism 373
11.1 Introduction 374
11.2 Superclasses and Subclasses 374
11.3 Using the super Keyword 380
11.4 Overriding Methods 382
11.5 Overriding vs. Overloading 383
11.6 The Object Class and Its toString() Method 384
11.7 Polymorphism 384
11.8 Dynamic Binding 385
11.9 Casting Objects and the instanceof Operator 387
11.10 The Object's equals() Method 389
11.11 The ArrayList Class 390
11.12 A Custom Stack Class 393
11.13 The protected Data and Methods 394
11.14 Preventing Extending and Overriding 396

Chapter 12 GUI Basics 405
12.1 Introduction 406
12.2 Swing vs. AWT 406
12.3 The Java GUI API 406
12.4 Frames 408
12.5 Layout Managers 411

12.6 Using Panels as Subcontainers 417
12.7 The Color Class 419
12.8 The Font Class 419
12.9 Common Features of Swing GUI Components 420
12.10 Image Icons 422

Chapter 13 Exception Handling 431
13.1 Introduction 432
13.2 Exception-Handling Overview 432
13.3 Exception-Handling Advantages 434
13.4 Exception Types 437
13.5 More on Exception Handling 439
13.6 The finally Clause 445
13.7 When to Use Exceptions 447
13.8 Rethrowing Exceptions 447
13.9 Chained Exceptions 447
13.10 Creating Custom Exception Classes 448

Chapter 14 Abstract Classes and Interfaces 457
14.1 Introduction 458
14.2 Abstract Classes 458
14.3 Example: Calendar and GregorianCalendar 462
14.4 Interfaces 465
14.5 Example: The Comparable Interface 467
14.6 Example: The ActionListener Interface 469
14.7 Example: The Cloneable Interface 471
14.8 Interfaces vs. Abstract Classes 473
14.9 Processing Primitive Data Type Values as Objects 476
14.10 Sorting an Array of Objects 479
14.11 Automatic Conversion between Primitive Types and Wrapper Class Types 481
14.12 The BigInteger and BigDecimal Classes 481
14.13 Case Study: The Rational Class 482

Chapter 15 Graphics 497
15.1 Introduction 498
15.2 Graphical Coordinate Systems 498
15.3 The Graphics Class 499
15.4 Drawing Strings, Lines, Rectangles, and Ovals 501
15.5 Case Study: The FigurePanel Class 502
15.6 Drawing Arcs 506

15.7 Drawing Polygons and Polylines 507
15.8 Centering a String Using the `FontMetrics` Class 510
15.9 Case Study: The `MessagePanel` Class 512
15.10 Case Study: The `StillClock` Class 516
15.11 Displaying Images 520
15.12 Case Study: The `ImageViewer` Class 522

Chapter 16 Event-Driven Programming 533
16.1 Introduction 534
16.2 Event and Event Source 534
16.3 Listeners, Registrations, and Handling Events 535
16.4 Inner Classes 541
16.5 Anonymous Class Listeners 542
16.6 Alternative Ways of Defining Listener Classes 544
16.7 Problem: Loan Calculator 547
16.8 Window Events 549
16.9 Listener Interface Adapters 551
16.10 Mouse Events 552
16.11 Key Events 555
16.12 Animation Using the `Timer` Class 557

Chapter 17 Creating Graphical User Interfaces 571
17.1 Introduction 572
17.2 Buttons 572
17.3 Check Boxes 578
17.4 Radio Buttons 581
17.5 Labels 583
17.6 Text Fields 584
17.7 Text Areas 586
17.8 Combo Boxes 590
17.9 Lists 593
17.10 Scroll Bars 596
17.11 Sliders 599
17.12 Creating Multiple Windows 602

Chapter 18 Applets and Multimedia 613
18.1 Introduction 614
18.2 Developing Applets 614
18.3 The HTML File and the `<applet>` Tag 615
18.4 Applet Security Restrictions 618
18.5 Enabling Applets to Run as Applications 618

18.6 Applet Life-Cycle Methods 620
18.7 Passing Strings to Applets 620
18.8 Case Study: Bouncing Ball 624
18.9 Case Study: TicTacToe 628
18.10 Locating Resources Using the URL Class 632
18.11 Playing Audio in Any Java Program 633
18.12 Case Study: Multimedia Animations 634

Chapter 19 Binary I/O 649

19.1 Introduction 650
19.2 How is I/O Handled in Java? 650
19.3 Text I/O vs. Binary I/O 650
19.4 Binary I/O Classes 652
19.5 Problem: Copying Files 660
19.6 Object I/O 662
19.7 Random-Access Files 666

Chapter 20 Recursion 677

20.1 Introduction 678
20.2 Problem: Computing Factorials 678
20.3 Problem: Computing Fibonacci Numbers 681
20.4 Problem Solving Using Recursion 683
20.5 Recursive Helper Methods 684
20.6 Problem: Finding the Directory Size 687
20.7 Problem: Towers of Hanoi 688
20.8 Problem: Fractals 692
20.9 Problem: Eight Queens 695
20.10 Recursion vs. Iteration 697
20.11 Tail Recursion 697

APPENDIXES

Appendix A Java Keywords 707
Appendix B The ASCII Character Set 710
Appendix C Operator Precedence Chart 712
Appendix D Java Modifiers 714
Appendix E Special Floating-Point Values 716
Appendix F Number Systems 717

INDEX 721

INTRODUCTION TO JAVA™ PROGRAMMING

This page intentionally left blank

CHAPTER 1

INTRODUCTION TO COMPUTERS, PROGRAMS, AND JAVA

Objectives

- To review computer basics, programs, and operating systems (§§1.2–1.4).
- To explore the relationship between Java and the World Wide Web (§1.5).
- To distinguish the terms API, IDE, and JDK (§1.6).
- To write a simple Java program (§1.7).
- To display output on the console (§1.7).
- To explain the basic syntax of a Java program (§1.7).
- To create, compile, and run Java programs (§1.8).
- (GUI) To display output using the `JOptionPane` output dialog boxes (§1.9).



1.1 Introduction

You use word processors to write documents, Web browsers to explore the Internet, and email programs to send email. These are all examples of software that runs on computers. Software is developed using programming languages. There are many programming languages—so *why Java*? The answer is that Java enables users to develop and deploy applications on the Internet for servers, desktop computers, and small hand-held devices. The future of computing is being profoundly influenced by the Internet, and Java promises to remain a big part of that future. Java is *the* Internet programming language.

why Java?

You are about to begin an exciting journey, learning a powerful programming language. At the outset, it is helpful to review computer basics, programs, and operating systems and to become familiar with number systems. If you are already familiar with such terms as CPU, memory, disks, operating systems, and programming languages, you may skip the review in §§1.2–1.4.

1.2 What Is a Computer?

A computer is an electronic device that stores and processes data. It includes both *hardware* and *software*. In general, hardware comprises the visible, physical elements of the computer, and software provides the invisible instructions that control the hardware and make it perform specific tasks. Writing instructions for computers to perform is called computer programming. Knowing computer hardware isn't essential to your learning a programming language, but it does help you understand better the effect of the program instructions. This section introduces computer hardware components and their functions.

hardware
software

A computer consists of the following major hardware components (Figure 1.1):

- Central processing unit (CPU)
- Memory (main memory)
- Storage devices (e.g., disks, CDs, tapes)
- Input and output devices (e.g., monitors, keyboards, mice, printers)
- Communication devices (e.g., modems and network interface cards (NICs))

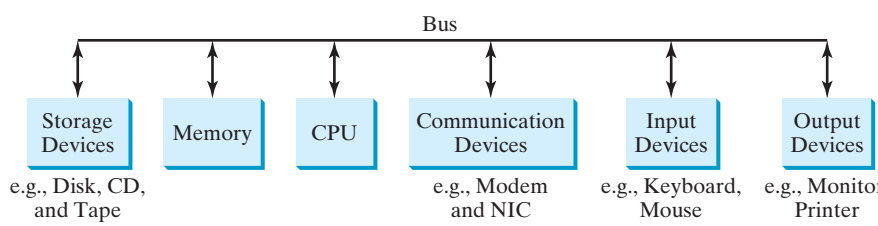


FIGURE 1.1 A computer consists of CPU, memory, storage devices, input devices, output devices, and communication devices.

bus

The components are connected through a subsystem called a *bus* that transfers data or power between them.

1.2.1 Central Processing Unit

CPU

The *central processing unit* (CPU) is the computer's brain. It retrieves instructions from memory and executes them. The CPU usually has two components: a *control unit* and an *arithmetic/logic unit*. The control unit controls and coordinates the actions of the other

components. The arithmetic/logic unit performs numeric operations (addition, subtraction, multiplication, division) and logical operations (comparisons).

Today's CPU is built on a small silicon semiconductor chip having millions of transistors. Every computer has an internal clock, which emits electronic pulses at a constant rate. These pulses are used to control and synchronize the pace of operations. The higher the clock speed, the more instructions are executed in a given period of time. The unit of measurement of clock speed is the *hertz* (*Hz*), with 1 hertz equaling 1 pulse per second. The clock speed of a computer is usually stated in megahertz (MHz) (1 MHz is 1 million Hz). CPU speed has been improved continuously. Intel's Pentium 3 Processor runs at about 500 megahertz and Pentium 4 Processor at about 3 gigahertz (GHz) (1 GHz is 1000 MHz).

speed
hertz
megahertz
gigahertz

1.2.2 Memory

To store and process information, computers use *off* and *on* electrical states, referred to by convention as *0* and *1*. These 0s and 1s are interpreted as digits in the binary number system and called *bits* (*b*inary dig*its*). Data of various kinds, such as numbers, characters, and strings, are encoded as series of bits. Data and program instructions for the CPU to execute are stored as groups of bits, or bytes, each byte composed of eight bits, in a computer's *memory*. A memory unit is an ordered sequence of *bytes*, as shown in Figure 1.2.

bit
byte

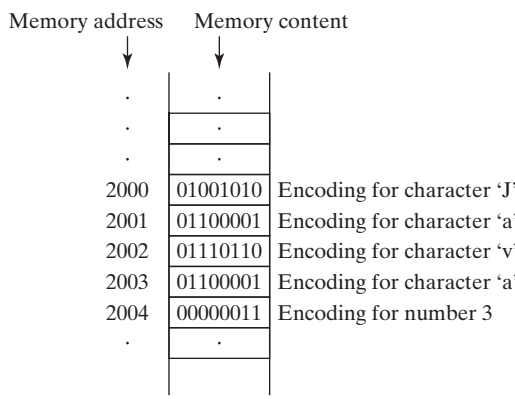

Memory address
Memory content
2000 01001010 Encoding for character 'J'
2001 01100001 Encoding for character 'a'
2002 01110110 Encoding for character 'v'
2003 01100001 Encoding for character 'a'
2004 00000011 Encoding for number 3

FIGURE 1.2 Memory stores data and program instructions.

The programmer need not be concerned about the encoding and decoding of data, which the system performs automatically, based on the encoding scheme. In the popular ASCII encoding scheme, for example, character `'J'` is represented by `01001010` in one byte.

A byte is the minimum storage unit. A small number such as `3` can be stored in a single byte. To store a number that cannot fit into a single byte, the computer uses several adjacent bytes. No two data items can share or split the same byte.

A memory byte is never empty, but its initial content may be meaningless to your program. The current content of a memory byte is lost whenever new information is placed in it.

A program and its data must be brought to memory before they can be executed.

Every byte has a unique address. The address is used to locate the byte for storing and retrieving data. Since bytes can be accessed in any order, the memory is also referred to as *random-access memory* (*RAM*). Today's personal computers usually have at least 1 gigabyte of RAM. Computer storage size is measured in bytes, kilobytes (KB), megabytes (MB), gigabytes (GB), and terabytes (TB). A *kilobyte* is $2^{10} = 1024$, about 1000 bytes, a *megabyte* is $2^{20} = 1048576$, about 1 million bytes, a *gigabyte* is about 1 billion bytes, and a terabyte is about 1000 gigabytes. Like the CPU, memory is built on silicon semiconductor chips having thousands of transistors embedded on their surface. Compared to CPU chips, memory chips are less complicated, slower, and less expensive.

RAM
megabyte

1.2.3 Storage Devices

Memory is volatile, because information is lost when the power is turned off. Programs and data are permanently stored on storage devices and are moved, when the computer actually uses them, to memory, which is much faster than storage devices.

There are four main types of storage devices:

- Disk drives
- CD drives (CD-R, CD-RW, and DVD)
- Tape drives
- USB flash drives

drive

Drives are devices for operating a medium, such as disks, CDs, and tapes.

Disks

hard disk

Each computer has at least one hard drive. *Hard disks* are for permanently storing data and programs. The hard disks of the latest PCs store from 80 to 250 gigabytes. Often disk drives are encased inside the computer. Removable hard disks are also available.

CDs and DVDs

CD-R

CD-RW

CD stands for compact disk. There are two types of CD drives: CD-R and CD-RW. A *CD-R* is for read-only permanent storage; the user cannot modify its contents once they are recorded. A *CD-RW* can be used like a hard disk and can be both read and rewritten. A single CD can hold up to 700 MB. Most software is distributed through CD-ROMs. Most new PCs are equipped with a CD-RW drive that can work with both CD-R and CD-RW.

DVD stands for digital versatile disc or digital video disk. DVDs and CDs look alike, and you can use either to store data. A DVD can hold more information than a CD. A standard DVD's storage capacity is 4.7 GB.

Tapes

Tapes are mainly used for backup of data and programs. Unlike disks and CDs, tapes store information sequentially. The computer must retrieve information in the order it was stored. Tapes are very slow. It would take one to two hours to back up a 1-gigabyte hard disk. The new trend is to back up data using flash drives or external hard disks.

USB Flash Drives

USB flash drives are devices for storing and transporting data. A flash drive is small—about the size of a pack of gum. It acts like a portable hard drive that can be plugged into your computer's USB port. USB flash drives are currently available with up to 32 GB storage capacity.

1.2.4 Input and Output Devices

Input and output devices let the user communicate with the computer. The common input devices are *keyboards* and *mice*. The common output devices are *monitors* and *printers*.

The Keyboard

A computer *keyboard* resembles a typewriter keyboard with extra keys added for certain special functions.

function key

Function keys are located at the top of the keyboard and are numbered with prefix F. Their use depends on the software.

modifier key

A *modifier key* is a special key (e.g., *Shift*, *Alt*, *Ctrl*) that modifies the normal action of another key when the two are pressed in combination.

The *numeric keypad,* located on the right-hand corner of the keyboard, is a separate set of keys for quick input of numbers. numeric keypad

Arrow keys, located between the main keypad and the numeric keypad, are used to move the cursor up, down, left, and right.

The *Insert, Delete, Page Up, and Page Down keys*, located above the arrow keys, are used in word processing for performing insert, delete, page up, and page down.

The Mouse

A *mouse* is a pointing device. It is used to move an electronic pointer called a cursor around the screen or to click on an object on the screen to trigger it to respond.

The Monitor

The *monitor* displays information (text and graphics). The screen resolution and dot pitch determine the quality of the display.

The *screen resolution* specifies the number of pixels per square inch. Pixels (short for "picture elements") are tiny dots that form an image on the screen. A common resolution for a 17-inch screen, for example, is 1024 pixels wide and 768 pixels high. The resolution can be set manually. The higher the resolution, the sharper and clearer the image is. screen resolution

The *dot pitch* is the amount of space between pixels in millimeters. The smaller the dot pitch, the better the display. dot pitch

1.2.5 Communication Devices

Computers can be networked through communication devices, such as the dialup modem (*mo*dulator/*dem*odulator), DSL, cable modem, network interface card, and wireless. A dialup modem uses a phone line and can transfer data at a speed up to 56,000 bps (bits per second). A *DSL* (digital subscriber line) also uses a phone line and can transfer data twenty times faster. A cable modem uses the TV cable line maintained by the cable company and is as fast as a DSL. A network interface card (NIC) is a device that connects a computer to a local area network (LAN). The *LAN* is commonly used in universities and business and government organizations. A typical *NIC* called *10BaseT* can transfer data at 10 *mbps* (million bits per second). Wireless is becoming popular. Every laptop sold today is equipped with a wireless adapter that enables the computer to connect with the Internet.

modem

DSL

NIC

LAN

mbps

1.3 Programs

Computer *programs*, known as *software*, are instructions to the computer, telling it what to do. Computers do not understand human languages, so you need to use computer languages in computer programs. *Programming* is the creation of a program that is executable by a computer and performs the required tasks.

software

programming

A computer's native language, which differs among different types of computers, is its *machine language*—a set of built-in primitive instructions. These instructions are in the form of binary code, so in telling the machine what to do, you have to enter binary code. Programming in machine language is a tedious process. Moreover, the programs are highly difficult to read and modify. For example, to add two numbers, you might have to write an instruction in binary like this:

machine language

```
1101101010011010
```

Assembly language is a low-level programming language in which a mnemonic is used to represent each of the machine-language instructions. For example, to add two numbers, you might write an instruction in assembly code like this:

assembly language

```
ADDF3 R1, R2, R3
```

assembler

Assembly languages were developed to make programming easy. However, since the computer cannot understand assembly language, a program called an *assembler* is used to convert assembly-language programs into machine code, as shown in Figure 1.3.

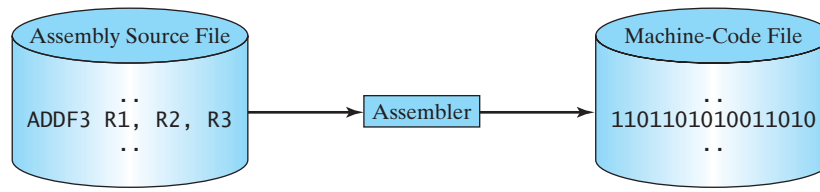


FIGURE 1.3 Assembler translates assembly-language instructions to machine code.

Assembly programs are written in terms of machine instructions with easy-to-remember mnemonic names. Since assembly language is machine dependent, an assembly program can be executed only on a particular kind of machine. The high-level languages were developed in order to transcend platform specificity and make programming easier.

high-level language

The *high-level languages* are English-like and easy to learn and program. Here, for example, is a high-level language statement that computes the area of a circle with radius 5:

```
area = 5 * 5 * 3.1415;
```

Among the more than one hundred high-level languages, the following are well known:

- COBOL (COmmon Business Oriented Language)
- FORTRAN (FORmula TRANslation)
- BASIC (Beginner's All-purpose Symbolic Instruction Code)
- Pascal (named for Blaise Pascal)
- Ada (named for Ada Lovelace)
- C (developed by the designer of B)
- Visual Basic (Basic-like visual language developed by Microsoft)
- Delphi (Pascal-like visual language developed by Borland)
- C++ (an object-oriented language, based on C)
- C# (a Java-like language developed by Microsoft)
- Java

Each of these languages was designed for a specific purpose. COBOL was designed for business applications and is used primarily for business data processing. FORTRAN was designed for mathematical computations and is used mainly for numeric computations. BASIC was designed to be learned and used easily. Ada was developed for the Department of Defense and is used mainly in defense projects. C combines the power of an assembly language with the ease of use and portability of a high-level language. Visual Basic and Delphi are used in developing graphical user interfaces and in rapid application development. C++ is popular for system software projects such as writing compilers and operating systems. The Microsoft Windows operating system was coded using C++. C# (pronounced C sharp) is a new language developed by Microsoft for developing applications based on the Microsoft .NET platform. Java, developed by Sun Microsystems, is widely used for developing platform-independent Internet applications.

A program written in a high-level language is called a *source program* or *source code*. Since a computer cannot understand a source program, a program called a *compiler* is used to translate it into a machine-language program. The machine-language program is then linked with other supporting library code to form an executable file, which can be run on the machine, as shown in Figure 1.4. On Windows, executable files have extension .exe.

source program
compiler

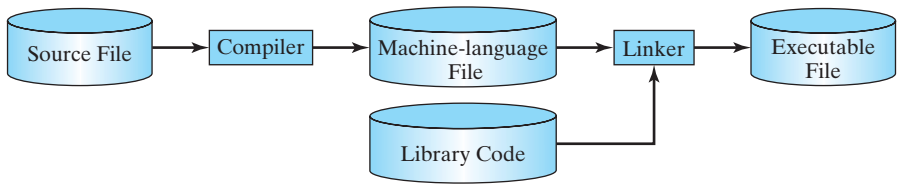


FIGURE 1.4 A source program is compiled into a machine-language file, which is then linked with the system library to form an executable file.

1.4 Operating Systems

The *operating system* (*OS*) is the most important program that runs on a computer, which manages and controls a computer's activities. The popular operating systems are Microsoft Windows, Mac OS, and Linux. Application programs, such as a Web browser or a word processor, cannot run without an operating system. The interrelationship of hardware, operating system, application software, and the user is shown in Figure 1.5.

OS

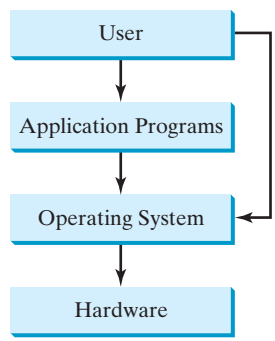


FIGURE 1.5 The operating system is the software that controls and manages the system.

The major tasks of an operating system are:

- Controlling and monitoring system activities
- Allocating and assigning system resources
- Scheduling operations

1.4.1 Controlling and Monitoring System Activities

Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the monitor, keeping track of files and directories on the disk, and controlling peripheral devices, such as disk drives and printers. They also make sure that different programs and users running at the same time do not interfere with each other, and they are responsible for security, ensuring that unauthorized users do not access the system.

1.4.2 Allocating and Assigning System Resources

The operating system is responsible for determining what computer resources a program needs (e.g., CPU, memory, disks, input and output devices) and for allocating and assigning them to run the program.

1.4.3 Scheduling Operations

The OS is responsible for scheduling programs to make efficient use of system resources. Many of today's operating systems support such techniques as *multiprogramming*, *multithreading*, or *multiprocessing* to increase system performance.

multiprogramming

Multiprogramming allows multiple programs to run simultaneously by sharing the CPU. The CPU is much faster than the computer's other components. As a result, it is idle most of the time—for example, while waiting for data to be transferred from the disk or from other sources. A multiprogramming OS takes advantage of this situation by allowing multiple programs to use the CPU when it would otherwise be idle. For example, you may use a word processor to edit a file at the same time as the Web browser is downloading a file.

multithreading

Multithreading allows concurrency within a program, so that its subtasks can run at the same time. For example, a word-processing program allows users to simultaneously edit text and save it to a file. In this example, editing and saving are two tasks within the same application. These two tasks may run on separate threads concurrently.

multiprocessing

Multiprocessing, or parallel processing, uses two or more processors together to perform a task. It is like a surgical operation where several doctors work together on one patient.

1.5 Java, World Wide Web, and Beyond

This book introduces Java programming. Java was developed by a team led by James Gosling at Sun Microsystems. Originally called *Oak*, it was designed in 1991 for use in embedded chips in consumer electronic appliances. In 1995, renamed *Java*, it was redesigned for developing Internet applications. For the history of Java, see java.sun.com/features/1998/05/birthday.html.

Java has become enormously popular. Its rapid rise and wide acceptance can be traced to its design characteristics, particularly its promise that you can write a program once and run it anywhere. As stated by Sun, Java is *simple*, *object oriented*, *distributed*, *interpreted*, *robust*, *secure*, *architecture neutral*, *portable*, *high performance*, *multithreaded*, and *dynamic*. For the anatomy of Java characteristics, see www.cs.armstrong.edu/liang/JavaCharacteristics.pdf.

Java is a full-featured, general-purpose programming language that can be used to develop robust mission-critical applications. Today, it is employed not only for Web programming, but also for developing standalone applications across platforms on servers, desktops, and mobile devices. It was used to develop the code to communicate with and control the robotic rover on Mars. Many companies that once considered Java to be more hype than substance are now using it to create distributed applications accessed by customers and partners across the Internet. For every new project being developed today, companies are asking how they can use Java to make their work easier.

The World Wide Web is an electronic information repository that can be accessed on the Internet from anywhere in the world. The Internet, the Web's infrastructure, has been around for more than thirty years. The colorful World Wide Web and sophisticated Web browsers are the major reason for the Internet's popularity.

The primary authoring language for the Web is the Hypertext Markup Language (HTML). HTML is a simple language for laying out documents, linking documents on the Internet, and bringing images, sound, and video alive on the Web. However, it cannot interact with the user except through simple forms. Web pages in HTML are essentially static and flat.

Java initially became attractive because Java programs can be run from a Web browser. Such programs are called *applets*. Applets employ a modern graphical interface with buttons,

applet

text fields, text areas, radio buttons, and so on, to interact with users on the Web and process their requests. Applets make the Web responsive, interactive, and fun to use. Figure 1.6 shows an applet running from a Web browser for playing a Tic Tac Toe game.

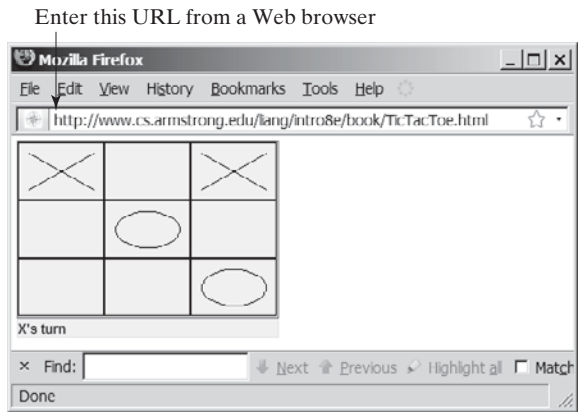


FIGURE 1.6 A Java applet for playing TicTacToe is embedded in an HTML page.



Tip

For a demonstration of Java applets, visit java.sun.com/applets. This site provides a rich Java resource as well as links to other cool applet demo sites. java.sun.com is the official Sun Java Website.

Java can also be used to develop applications on the server side. These applications can be run from a Web server to generate dynamic Web pages. The automatic grading system for this book, as shown in Figure 1.7, was developed using Java.

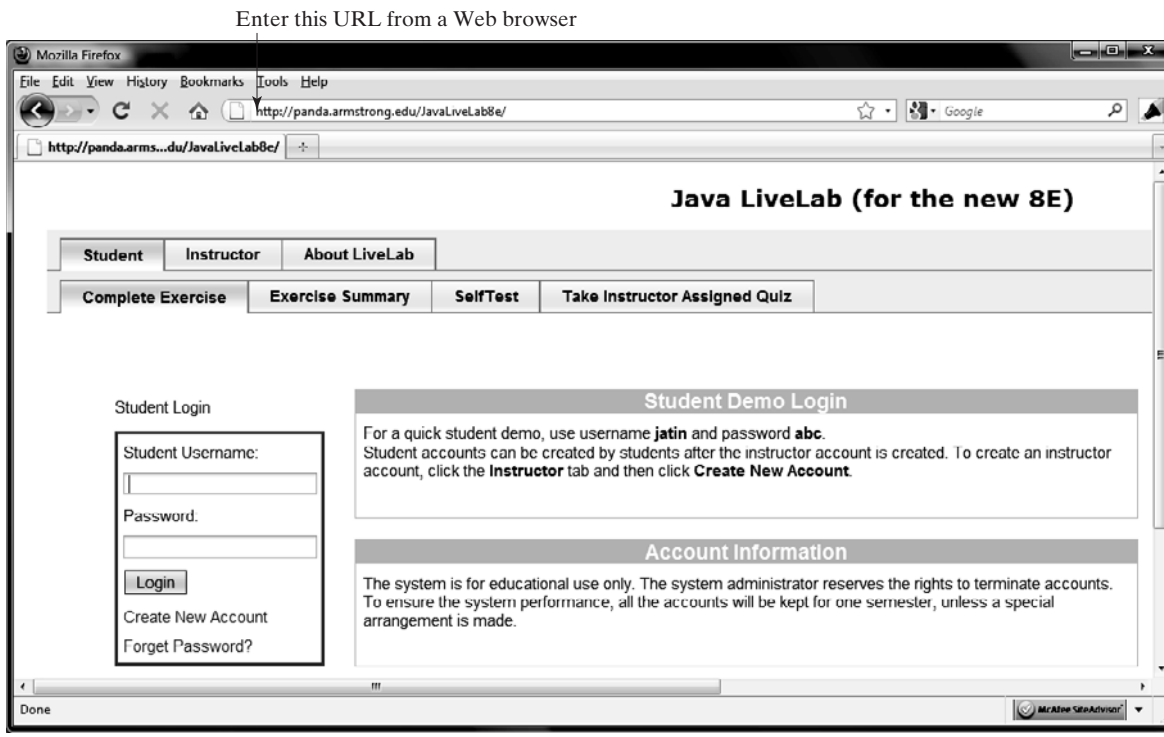

Enter this URL from a Web browser
Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://panda.armstrong.edu/JavaLiveLab8e/
Java LiveLab (for the new 8E)
Student
Instructor
About LiveLab
Complete Exercise
Exercise Summary
SelfTest
Take Instructor Assigned Quiz
Student Login
Student Username:
Password:
Login
Create New Account
Forget Password?
Student Demo Login
For a quick student demo, use username jatin and password abc.
Student accounts can be created by students after the instructor account is created. To create an instructor account, click the Instructor tab and then click Create New Account.
Account Information
The system is for educational use only. The system administrator reserves the rights to terminate accounts. To ensure the system performance, all the accounts will be kept for one semester, unless a special arrangement is made.
Done

FIGURE 1.7 Java was used to develop an automatic grading system to accompany this book.

Java is a versatile programming language. You can use it to develop applications on your desktop and on the server. You can also use it to develop applications for small hand-held devices. Figure 1.8 shows a Java-programmed calendar displayed on a BlackBerry© and on a cell phone.

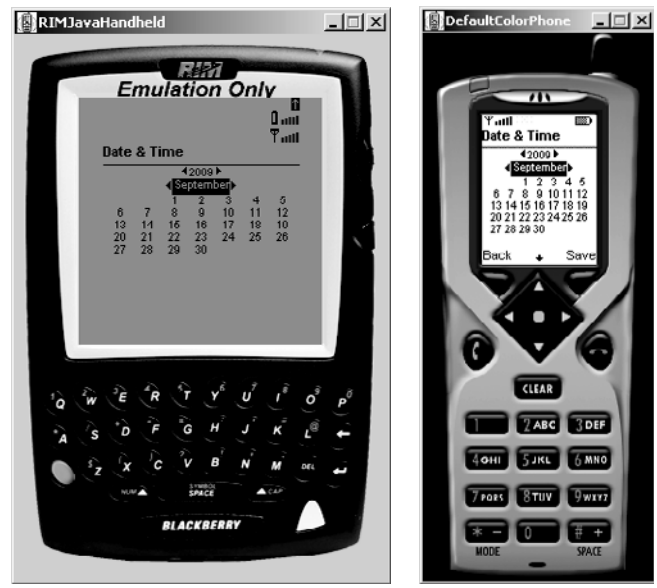


FIGURE 1.8 Java can be used to develop applications for hand-held and wireless devices, such as a BlackBerry© (left) and a cell phone (right).

1.6 The Java Language Specification, API, JDK, and IDE

Computer languages have strict rules of usage. If you do not follow the rules when writing a program, the computer will be unable to understand it. The Java language specification and Java API define the Java standard.

Java language specification

The *Java language specification* is a technical definition of the language that includes the syntax and semantics of the Java programming language. The complete Java language specification can be found at java.sun.com/docs/books/jls.

API

The *application program interface* (*API*) contains predefined classes and interfaces for developing Java programs. The Java language specification is stable, but the API is still expanding. At the Sun Java Website (java.sun.com), you can view and download the latest version of the Java API.

Java SE, EE, and ME

Java is a full-fledged and powerful language that can be used in many ways. It comes in three editions: *Java Standard Edition* (*Java SE*), *Java Enterprise Edition* (*Java EE*), and *Java Micro Edition* (*Java ME*). Java SE can be used to develop client-side standalone applications or applets. Java EE can be used to develop server-side applications, such as Java servlets and JavaServer Pages. Java ME can be used to develop applications for mobile devices, such as cell phones. This book uses Java SE to introduce Java programming.

There are many versions of Java SE. The latest, Java SE 6, will be used in this book. Sun releases each version with a *Java Development Toolkit* (*JDK*). For Java SE 6, the Java Development Toolkit is called *JDK 1.6* (also known as *Java 6 or JDK 6*).

JDK 1.6 = JDK 6

JDK consists of a set of separate programs, each invoked from a command line, for developing and testing Java programs. Besides JDK, you can use a Java development tool (e.g., NetBeans, Eclipse, and TextPad)—software that provides an *integrated development environment* (*IDE*) for rapidly developing Java programs. Editing, compiling, building, debugging, and

Java IDE

online help are integrated in one graphical user interface. Just enter source code in one window or open an existing file in a window, then click a button, menu item, or function key to compile and run the program.

1.7 A Simple Java Program

Let us begin with a simple Java program that displays the message "Welcome to Java!" on the console. *Console* refers to text entry and display device of a computer. The program is shown in Listing 1.1.

console



Video Note
First Java program

LISTING 1.1 Welcome.java

```
1 public class Welcome {                                    class
2   public static void main(String[] args) {                main method
3     // Display message Welcome to Java! to the console
4     System.out.println("Welcome to Java!");                display message
5   }
6 }
```

```
Welcome to Java!
```



The *line numbers* are displayed for reference purposes but are not part of the program. So, don't type line numbers in your program.

line numbers

Line 1 defines a class. Every Java program must have at least one class. Each class has a name. By convention, class names start with an uppercase letter. In this example, the *class name* is `Welcome`.

class name

Line 2 defines the *main method*. In order to run a class, the class must contain a method named `main`. The program is executed from the `main` method.

main method

A method is a construct that contains statements. The `main` method in this program contains the `System.out.println` statement. This statement prints a message `"Welcome to Java!"` to the console (line 4). Every statement in Java ends with a semicolon (`;`), known as the *statement terminator*.

statement terminator

Reserved words, or *keywords,* have a specific meaning to the compiler and cannot be used for other purposes in the program. For example, when the compiler sees the word `class`, it understands that the word after `class` is the name for the class. Other reserved words in this program are `public`, `static`, and `void`.

reserved word

Line 3 is a *comment* that documents what the program is and how it is constructed. Comments help programmers to communicate and understand the program. They are not programming statements and thus are ignored by the compiler. In Java, comments are preceded by two slashes (`//`) on a line, called a *line comment*, or enclosed between `/*` and `*/` on one or several lines, called a *block comment*. When the compiler sees `//`, it ignores all text after `//` on the same line. When it sees `/*`, it scans for the next `*/` and ignores any text between `/*` and `*/`. Here are examples of comments:

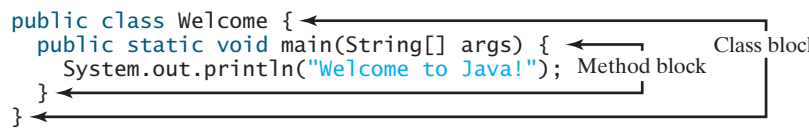
comment

```
// This application program prints Welcome to Java!
/* This application program prints Welcome to Java! */
/* This application program
   prints Welcome to Java! */
```

A pair of braces in a program forms a *block* that groups the program's components. In Java, each block begins with an opening brace (`{`) and ends with a closing brace (`}`). Every class has a *class block* that groups the data and methods of the class. Every method has a *method*

block

block that groups the statements in the method. Blocks can be *nested*, meaning that one block can be placed within another, as shown in the following code.



matching braces

Tip

An opening brace must be matched by a closing brace. Whenever you type an opening brace, immediately type a closing brace to prevent the missing-brace error. Most Java IDEs automatically insert the closing brace for each opening brace.

Note

You are probably wondering why the `main` method is declared this way and why `System.out.println(...)` is used to display a message to the console. For the time being, simply accept that this is how things are done. Your questions will be fully answered in subsequent chapters.

case sensitive

Caution

Java source programs are case sensitive. It would be wrong, for example, to replace `main` in the program with `Main`.

syntax rules

Note

Like any programming language, Java has its own syntax, and you need to write code that obeys the *syntax rules*. If your program violates the rules—for example if the semicolon is missing, a brace is missing, a quotation mark is missing, or `String` is misspelled—the Java compiler will report syntax errors. Try to compile the program with these errors and see what the compiler reports.

The program in Listing 1.1 displays one message. Once you understand the program, it is easy to extend it to display more messages. For example, you can rewrite the program to display three messages, as shown in Listing 1.2.

LISTING 1.2 Welcome1.java

class
main method
display message

```
1 public class Welcome1 {
2   public static void main(String[] args) {
3     System.out.println("Programming is fun!");
4     System.out.println("Fundamentals First");
5     System.out.println("Problem Driven");
6   }
7 }
```

```
Programming is fun!
Fundamentals First
Problem Driven
```

Further, you can perform mathematical computations and display the result to the console. Listing 1.3 gives an example of evaluating $\frac{10.5 + 2 \times 3}{45 - 3.5}$.

LISTING 1.3 ComputeExpression.java

```
1 public class ComputeExpression {                          class
2   public static void main(String[] args) {                main method
3     System.out.println((10.5 + 2 * 3) / (45 - 3.5));      compute expression
4   }
5 }
```

```
0.39759036144578314
```



The multiplication operator in Java is *. As you see, it is a straightforward process to translate an arithmetic expression to a Java expression. We will discuss Java expressions further in Chapter 2.

1.8 Creating, Compiling, and Executing a Java Program

You have to create your program and compile it before it can be executed. This process is repetitive, as shown in Figure 1.9. If your program has compilation errors, you have to modify the program to fix them, then recompile it. If your program has runtime errors or does not produce the correct result, you have to modify the program, recompile it, and execute it again.

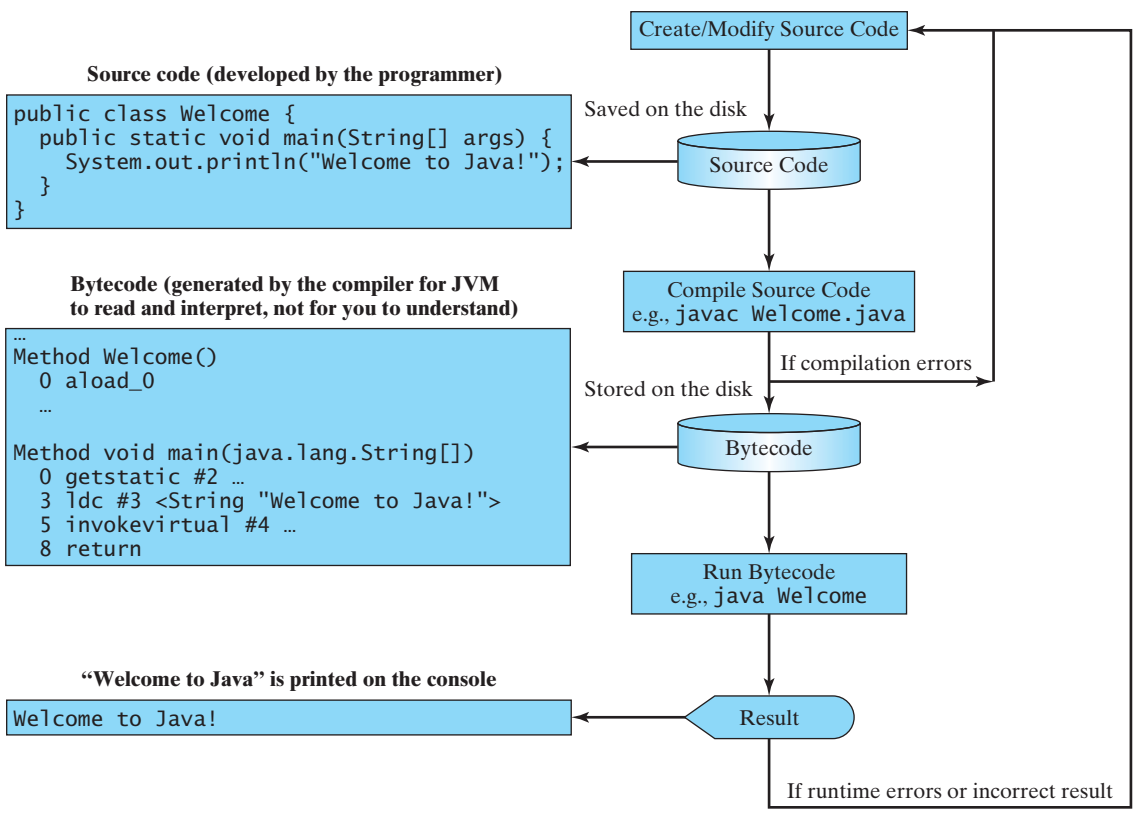


FIGURE 1.9 The Java program-development process consists of repeatedly creating/modifying source code, compiling, and executing programs.

editor

You can use any text *editor* or IDE to create and edit a Java source-code file. This section demonstrates how to create, compile, and run Java programs from a command window. If you wish to use an IDE such as Eclipse, NetBeans, or TextPad, please refer to Supplement II for tutorials. From the command window, you can use the NotePad to create the Java source code file, as shown in Figure 1.10.



Video Note
Brief Eclipse Tutorial

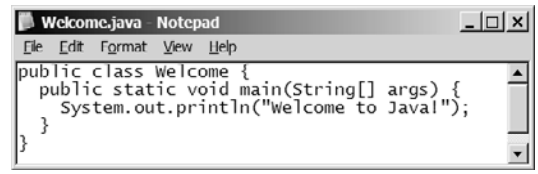


FIGURE 1.10 You can create the Java source file using Windows NotePad.



Note

file name

The source file must end with the extension .java and must have exactly the same name as the public class name. For example, the file for the source code in Listing 1.1 should be named Welcome.java, since the public class name is `Welcome`.

A Java compiler translates a Java source file into a Java bytecode file. The following command compiles Welcome.java:

compile

```
javac Welcome.java
```



Note

Supplement I.B

You must first install and configure JDK before compiling and running programs. See Supplement I.B, "Installing and Configuring JDK 6," on how to install JDK and set up the environment to compile and run Java programs. If you have trouble compiling and running programs, please see

Supplement I.C

Supplement I.C, "Compiling and Running Java from the Command Window." This supplement also explains how to use basic DOS commands and how to use Windows NotePad and WordPad to create and edit files. All the supplements are accessible from the Companion Website.

.class bytecode file

If there are no syntax errors, the *compiler* generates a bytecode file with a .class extension. So the preceding command generates a file named **Welcome. class**, as shown in Figure 1.11(a). The Java language is a high-level language while Java bytecode is a low-level language. The bytecode is similar to machine instructions but is architecture neutral and can run on any platform that has a Java Virtual Machine (JVM), as shown in Figure 1.11(b). Rather than a physical machine, the virtual machine is a program that interprets Java bytecode. This is one of Java's primary advantages: *Java bytecode can run on a variety of hardware platforms and operating systems.*

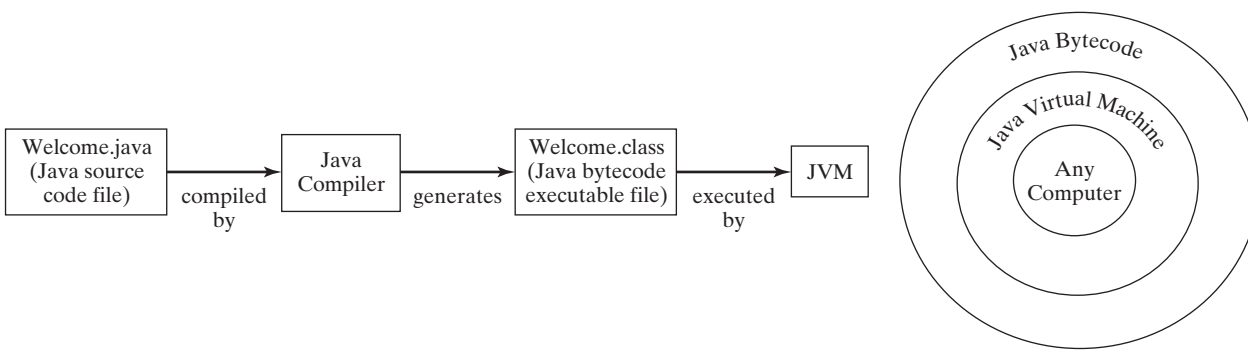


FIGURE 1.11 (a) Java source code is translated into bytecode. (b) Java bytecode can be executed on any computer with a Java Virtual Machine.

To execute a Java program is to run the program's bytecode. You can execute the bytecode on any platform with a JVM. Java bytecode is interpreted. Interpreting translates the individual steps in the bytecode into the target machine-language code one at a time rather than translating the whole program as a single unit. Each step is executed immediately after it is translated.

interpreting bytecode

The following command runs the bytecode:

```
java Welcome
```

run

Figure 1.12 shows the **javac** command for compiling Welcome.java. The compiler generated the Welcome.class file. This file is executed using the **java** command.

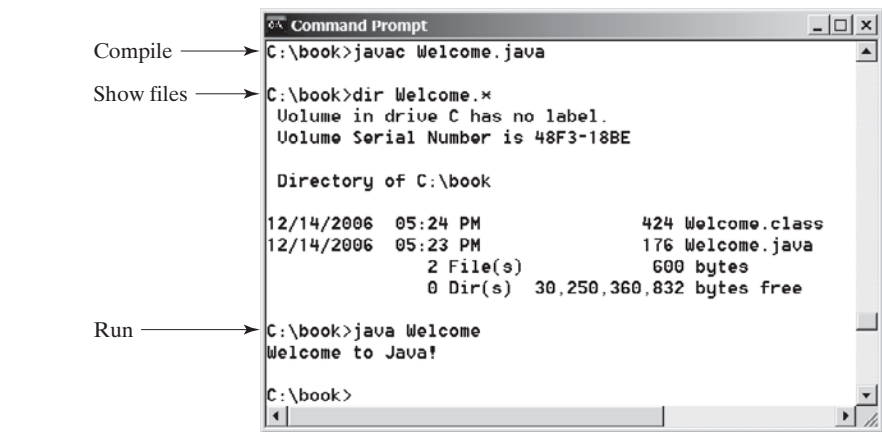


FIGURE 1.12 The output of Listing 1.1 displays the message "Welcome to Java!"



Video Note
Compile and run a Java program



Note

For simplicity and consistency, all source code and class files are placed under **c:\book** unless specified otherwise.

c:\book



Caution

Do not use the extension .class in the command line when executing the program. Use **java ClassName** to run the program. If you use **java ClassName.class** in the command line, the system will attempt to fetch **ClassName.class.class**.

java ClassName



Tip

If you execute a class file that does not exist, a `NoClassDefFoundError` will occur. If you execute a class file that does not have a `main` method or you mistype the `main` method (e.g., by typing `Main` instead of `main`), a `NoSuchMethodError` will occur.

`NoClassDefFoundError`

`NoSuchMethodError`



Note

When executing a Java program, the JVM first loads the bytecode of the class to memory using a program called the *class loader*. If your program uses other classes, the class loader dynamically loads them just before they are needed. After a class is loaded, the JVM uses a program called *bytecode verifier* to check the validity of the bytecode and ensure that the bytecode does not violate Java's security restrictions. Java enforces strict security to make sure that Java programs arriving from the network do not harm your computer.

class loader

bytecode verifier



Pedagogical Note

Instructors may require students to use packages for organizing programs. For example, you may place all programs in this chapter in a package named *chapter1*. For instructions on how to use packages, please see Supplement I.F, "Using Packages to Organize the Classes in the Text."

using package

1.9 (GUI) Displaying Text in a Message Dialog Box

The program in Listing 1.1 displays the text on the console, as shown in Figure 1.12. You can rewrite the program to display the text in a message dialog box. To do so, you need to use the `showMessageDialog` method in the `JOptionPane` class. `JOptionPane` is one of the many predefined classes in the Java system that you can reuse rather than "reinventing the wheel." You can use the `showMessageDialog` method to display any text in a message dialog box, as shown in Figure 1.13. The new program is given in Listing 1.4.

`JOptionPane`

`showMessageDialog`

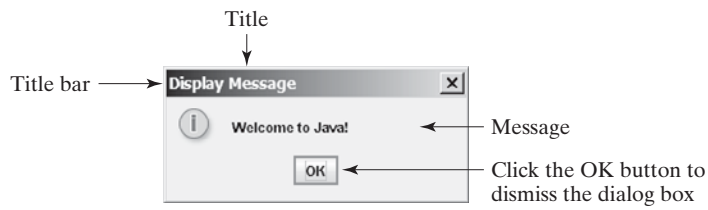


FIGURE 1.13 "Welcome to Java!" is displayed in a message box.

LISTING 1.4 WelcomeInMessageDialogBox.java

block comment

import

main method

display message

```
1 /*  This application program displays Welcome to Java!
2  *  in a message dialog box.
3  */
4 import javax.swing.JOptionPane;
5
6 public class WelcomeInMessageDialogBox {
7   public static void main(String[] args) {
8     // Display Welcome to Java! in a message dialog box
9     JOptionPane.showMessageDialog(null, "Welcome to Java!");
10  }
11 }
```

package

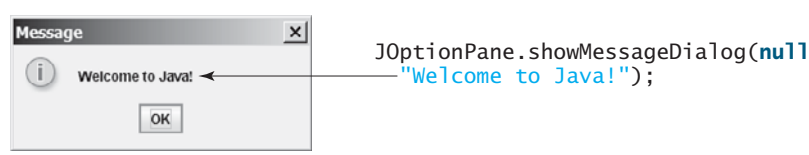
This program uses a Java class `JOptionPane` (line 9). Java's predefined classes are grouped into packages. `JOptionPane` is in the `javax.swing` package. `JOptionPane` is imported to the program using the `import` statement in line 4 so that the compiler can locate the class without the full name `javax.swing.JOptionPane`.



Note

If you replace `JOptionPane` on line 9 with `javax.swing.JOptionPane`, you don't need to import it in line 4. `javax.swing.JOptionPane` is the full name for the `JOptionPane` class.

The `showMessageDialog` method is a *static* method. Such a method should be invoked by using the class name followed by a dot operator (`.`) and the method name with arguments. Methods will be introduced in Chapter 5, "Methods." The `showMessageDialog` method can be invoked with two arguments, as shown below.



The first argument can always be `null`. `null` is a Java keyword that will be fully introduced in Chapter 8, "Objects and Classes." The second argument is a string for text to be displayed.

There are several ways to use the `showMessageDialog` method. For the time being, you need to know only two ways. One is to use a statement, as shown in the example:

two versions of `showMessageDialog`

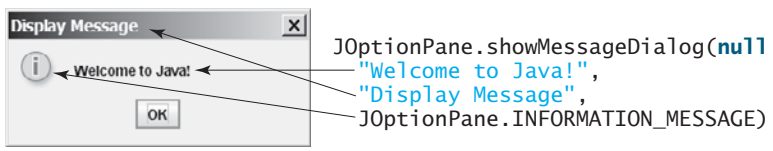
```
JOptionPane.showMessageDialog(null, x);
```

where `x` is a string for the text to be displayed.

The other is to use a statement like this one:

```
JOptionPane.showMessageDialog(null, x,
  y, JOptionPane.INFORMATION_MESSAGE);
```

where `x` is a string for the text to be displayed, and `y` is a string for the title of the message box. The fourth argument can be `JOptionPane.INFORMATION_MESSAGE`, which causes the icon ( ) to be displayed in the message box, as shown in the following example.





Note

There are two types of `import` statements: *specific import* and *wildcard import*. The *specific import* specifies a single class in the import statement. For example, the following statement imports `JOptionPane` from package `javax.swing`.

specific import

```
import javax.swing.JOptionPane;
```

The *wildcard import* imports all the classes in a package. For example, the following statement imports all classes from package `javax.swing`.

wildcard import

```
import javax.swing.*;
```

The information for the classes in an imported package is not read in at compile time or runtime unless the class is used in the program. The import statement simply tells the compiler where to locate the classes. There is no performance difference between a specific import and a wildcard import declaration.

no performance difference



Note

Recall that you have used the `System` class in the statement `System.out.println("Welcome to Java");` in Listing 1.1. The `System` class is not imported because it is in the `java.lang` package. All the classes in the `java.lang` package are *implicitly* imported in every Java program.

`java.lang` implicitly imported

Key Terms

.class file 14
.java file 14
assembly language 5
bit 3
block 12
block comment 11
bus 2
byte 3
bytecode 14
bytecode verifier 15
cable modem 5
central processing unit (CPU) 2
class loader 15
comment 11

compiler 7
console 11
dot pitch 5
DSL (digital subscriber line) 5
hardware 2
high-level language 6
Integrated Development Environment (IDE) 10
java command 15
javac command 15
Java Development Toolkit (JDK) 10
Java Virtual Machine (JVM) 14
keyword (or reserved word) 11
line comment 11
machine language 5
main method 11
memory 3
modem 5
network interface card (NIC) 5
operating system (OS) 7
pixel 5
program 5
programming 5
resolution 5
software 5
source code 7
source file 14
specific import 17
storage devices 4
statement 11
wildcard import 17



Note

Supplement I.A

The above terms are defined in the present chapter. Supplement I.A, "Glossary," lists all the key terms and descriptions in the book, organized by chapters.

Chapter Summary

1. A computer is an electronic device that stores and processes data.
2. A computer includes both *hardware* and *software*.
3. Hardware is the physical aspect of the computer that can be seen.
4. Computer *programs*, known as *software*, are the invisible instructions that control the hardware and make it perform tasks.
5. Computer programming is the writing of instructions (i.e., code) for computers to perform.
6. The central processing unit (CPU) is a computer's brain. It retrieves instructions from memory and executes them.
7. Computers use zeros and ones because digital devices have two stable states, referred to by convention as zero and one.
8. A bit is a binary digit 0 or 1.
9. A byte is a sequence of 8 bits.
10. A kilobyte is about 1000 bytes, a megabyte about 1 million bytes, a gigabyte about 1 billion bytes, and a terabyte about 1000 gigabytes.
11. Memory stores data and program instructions for the CPU to execute.
12. A memory unit is an ordered sequence of bytes.
13. Memory is volatile, because information is lost when the power is turned off.

14. Programs and data are permanently stored on storage devices and are moved to memory when the computer actually uses them.

15. The machine language is a set of primitive instructions built into every computer.

16. Assembly language is a low-level programming language in which a mnemonic is used to represent each machine-language instruction.

17. High-level languages are English-like and easy to learn and program.

18. A program written in a high-level language is called a source program.

19. A compiler is a software program that translates the source program into a machine-language program.

20. The operating system (OS) is a program that manages and controls a computer's activities.

21. Java is platform independent, meaning that you can write a program once and run it anywhere.

22. Java programs can be embedded in HTML pages and downloaded by Web browsers to bring live animation and interaction to Web clients.

23. Java source files end with the .java extension.

24. Every class is compiled into a separate bytecode file that has the same name as the class and ends with the .class extension.

25. To compile a Java source-code file from the command line, use the **javac** command.

26. To run a Java class from the command line, use the **java** command.

27. Every Java program is a set of class definitions. The keyword `class` introduces a class definition. The contents of the class are included in a block.

28. A block begins with an opening brace ({) and ends with a closing brace (}). Methods are contained in a class.

29. A Java program must have a `main` method. The `main` method is the entry point where the program starts when it is executed.

30. Every statement in Java ends with a semicolon (`;`), known as the *statement terminator.*

31. *Reserved words,* or *keywords,* have a specific meaning to the compiler and cannot be used for other purposes in the program.

32. In Java, comments are preceded by two slashes (`//`) on a line, called a *line comment*, or enclosed between `/*` and `*/` on one or several lines, called a block comment.

33. Java source programs are case sensitive.

34. There are two types of `import` statements: *specific import* and *wildcard import*. The *specific import* specifies a single class in the import statement. The *wildcard import* imports all the classes in a package.

Review Questions



Note

Answers to review questions are on the Companion Website.

Sections 1.2–1.4

1.1 Define hardware and software.

1.2 List the main components of the computer.

1.3 Define machine language, assembly language, and high-level programming language.

1.4 What is a source program? What is a compiler?

1.5 What is the JVM?

1.6 What is an operating system?

Sections 1.5–1.6

1.7 Describe the history of Java. Can Java run on any machine? What is needed to run Java on a computer?

1.8 What are the input and output of a Java compiler?

1.9 List some Java development tools. Are tools like NetBeans and Eclipse different languages from Java, or are they dialects or extensions of Java?

1.10 What is the relationship between Java and HTML?

Sections 1.7–1.9

1.11 Explain the Java keywords. List some Java keywords you learned in this chapter.

1.12 Is Java case sensitive? What is the case for Java keywords?

1.13 What is the Java source filename extension, and what is the Java bytecode filename extension?

1.14 What is a comment? Is the comment ignored by the compiler? How do you denote a comment line and a comment paragraph?

1.15 What is the statement to display a string on the console? What is the statement to display the message "Hello world" in a message dialog box?

1.16 The following program is wrong. Reorder the lines so that the program displays `morning` followed by `afternoon`.

```
  public static void main(String[] args) {
  }

public class Welcome {
    System.out.println("afternoon");
    System.out.println("morning");
}
```

1.17 Identify and fix the errors in the following code:

```
1 public class Welcome {
2   public void Main(String[] args) {
3     System.out.println('Welcome to Java!');
4   }
5 )
```

1.18 What is the command to compile a Java program? What is the command to run a Java program?

1.19 If a `NoClassDefFoundError` occurs when you run a program, what is the cause of the error?

1.20 If a `NoSuchMethodError` occurs when you run a program, what is the cause of the error?

1.21 Why does the `System` class not need to be imported?

1.22 Are there any performance differences between the following two `import` statements?

```
import javax.swing.JOptionPane;

import javax.swing.*;
```

1.23 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    System.out.println("3.5 * 4 / 2 - 2.5 is ");
    System.out.println(3.5 * 4 / 2 - 2.5);
  }
}
```

Programming Exercises



Note

Solutions to even-numbered exercises are on the Companion Website. Solutions to all exercises are on the Instructor Resource Website. The level of difficulty is rated easy (no star), moderate (*), hard (**), or challenging (***).

level of difficulty

1.1 (*Displaying three messages*) Write a program that displays `Welcome to Java`, `Welcome to Computer Science`, and `Programming is fun`.

1.2 (*Displaying five messages*) Write a program that displays `Welcome to Java` five times.

1.3* (*Displaying a pattern*) Write a program that displays the following pattern:

```
    J      A      V      V     A
    J     A A      V    V     A A
J   J    AAAAA      V V     AAAAA
 J J    A     A      V     A     A
```

1.4 (*Printing a table*) Write a program that displays the following table:

```
a       a^2     a^3
1       1       1
2       4       8
3       9       27
4       16      64
```

1.5 (*Computing expressions*) Write a program that displays the result of $\frac{9.5 \times 4.5 - 2.5 \times 3}{45.5 - 3.5}$.

1.6 (*Summation of a series*) Write a program that displays the result of $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$.

1.7 (*Approximating* π) π can be computed using the following formula:

$$\pi = 4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} + \cdots\right)$$

Write a program that displays the result of $4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13}\right)$. Use `1.0` instead of `1` in your program.

CHAPTER 2

ELEMENTARY PROGRAMMING

Objectives

- To write Java programs to perform simple calculations (§2.2).
- To obtain input from the console using the `Scanner` class (§2.3).
- To use identifiers to name variables, constants, methods, and classes (§2.4).
- To use variables to store data (§§2.5–2.6).
- To program with assignment statements and assignment expressions (§2.6).
- To use constants to store permanent data (§2.7).
- To declare Java primitive data types: `byte`, `short`, `int`, `long`, `float`, `double`, and `char` (§2.8.1).
- To use Java operators to write numeric expressions (§§2.8.2–2.8.3).
- To display the current time (§2.9).
- To use shorthand operators (§2.10).
- To cast the value of one type to another type (§2.11).
- To compute loan payments (§2.12).
- To represent characters using the `char` type (§2.13).
- To compute monetary changes (§2.14).
- To represent a string using the `String` type (§2.15).
- To become familiar with Java documentation, programming style, and naming conventions (§2.16).
- To distinguish syntax errors, runtime errors, and logic errors and debug errors (§2.17).
- (GUI) To obtain input using the `JOptionPane` input dialog boxes (§2.18).



2.1 Introduction

In Chapter 1 you learned how to create, compile, and run a Java program. Now you will learn how to solve practical problems programmatically. Through these problems, you will learn elementary programming using primitive data types, variables, constants, operators, expressions, and input and output.

2.2 Writing Simple Programs

problem

To begin, let's look at a simple problem for computing the area of a circle. How do we write a program for solving this problem?

algorithm

Writing a program involves designing algorithms and translating algorithms into code. An *algorithm* describes how a problem is solved in terms of the actions to be executed and the order of their execution. Algorithms can help the programmer plan a program before writing it in a programming language. Algorithms can be described in natural languages or in

pseudocode

pseudocode (i.e., natural language mixed with programming code). The algorithm for this program can be described as follows:

1. Read in the radius.
2. Compute the area using the following formula:

$$area = radius \times radius \times \pi$$

3. Display the area.

Many of the problems you will encounter when taking an introductory course in programming can be described with simple, straightforward algorithms.

When you *code*, you translate an algorithm into a program. You already know that every Java program begins with a class declaration in which the keyword `class` is followed by the class name. Assume that you have chosen `ComputeArea` as the class name. The outline of the program would look like this:

```
public class ComputeArea {
  // Details to be given later
}
```

As you know, every Java program must have a `main` method where program execution begins. So the program is expanded as follows:

```
public class ComputeArea {
  public static void main(String[] args) {
    // Step 1: Read in radius

    // Step 2: Compute area

    // Step 3: Display the area
  }
}
```

The program needs to read the radius entered by the user from the keyboard. This raises two important issues:

- Reading the radius.
- Storing the radius in the program.

variable

Let's address the second issue first. In order to store the radius, the program needs to declare a symbol called a *variable*. A variable designates a location in memory for storing data and computational results in the program. A variable has a name that can be used to access the memory location.

descriptive names

floating-point number

primitive data types

Rather than using `x` and `y` as variable names, choose descriptive names: in this case, `radius` for radius, and `area` for area. To let the compiler know what `radius` and `area` are, specify their data types. Java provides simple data types for representing integers, floating-point numbers (i.e., numbers with a decimal point), characters, and Boolean types. These types are known as *primitive data types* or *fundamental types*.

Declare `radius` and `area` as double-precision floating-point numbers. The program can be expanded as follows:

```
public class ComputeArea {
  public static void main(String[] args) {
    double radius;
    double area;

    // Step 1: Read in radius

    // Step 2: Compute area

    // Step 3: Display the area
  }
}
```

The program declares `radius` and `area` as variables. The reserved word `double` indicates that `radius` and `area` are double-precision floating-point values stored in the computer.

The first step is to read in `radius`. Reading a number from the keyboard is not a simple matter. For the time being, let us assign a fixed value to `radius` in the program.

The second step is to compute `area` by assigning the result of the expression `radius * radius * 3.14159` to `area`.

In the final step, display `area` on the console by using the `System.out.println` method.

The complete program is shown in Listing 2.1. A sample run of the program is shown in Figure 2.1.

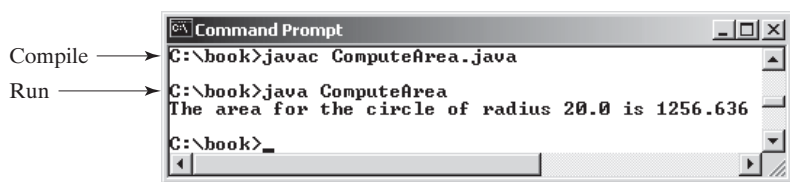


FIGURE 2.1 The program displays the area of a circle.

LISTING 2.1 ComputeArea.java

```
1 public class ComputeArea {
2   public static void main(String[] args) {
3     double radius; // Declare radius
4     double area; // Declare area
5
6     // Assign a radius
7     radius = 20; // New value is radius
8
9     // Compute area
10    area = radius * radius * 3.14159;
11
12    // Display results
```

```
13     System.out.println("The area for the circle of radius " +
14       radius + " is " + area);
15   }
16 }
```

declaring variable

assign value

Variables such as `radius` and `area` correspond to memory locations. Every variable has a name, a type, a size, and a value. Line 3 declares that `radius` can store a `double` value. The value is not defined until you assign a value. Line 7 assigns `20` into `radius`. Similarly, line 4 declares variable `area`, and line 10 assigns a value into `area`. The following table shows the value in the memory for `area` and `radius` as the program is executed. Each row in the table shows the values of variables after the statement in the corresponding line in the program is executed. Hand trace is helpful to understand how a program works, and it is also a useful tool for finding errors in the program.



line#	radius	area
3	no value	
4		no value
7	20	
10		1256.636

concatenating strings

concatenating strings with numbers

The plus sign (+) has two meanings: one for addition and the other for concatenating strings. The plus sign (+) in lines 13–14 is called a *string concatenation operator*. It combines two strings if two operands are strings. If one of the operands is a nonstring (e.g., a number), the nonstring value is converted into a string and concatenated with the other string. So the plus signs (+) in lines 13–14 concatenate strings into a longer string, which is then displayed in the output. Strings and string concatenation will be discussed further in §2.15, "The `String` Type."



Caution

breaking a long string

A string constant cannot cross lines in the source code. Thus the following statement would result in a compile error:

```
System.out.println("Introduction to Java Programming,
  by Y. Daniel Liang");
```

To fix the error, break the string into separate substrings, and use the concatenation operator (+) to combine them:

```
System.out.println("Introduction to Java Programming, " +
  "by Y. Daniel Liang");
```



Tip

incremental development and testing

This example consists of three steps. It is a good approach to develop and test these steps incrementally by adding them one at a time.

2.3 Reading Input from the Console



Video Note
Obtain input

In Listing 2.1, the radius is fixed in the source code. To use a different radius, you have to modify the source code and recompile it. Obviously, this is not convenient. You can use the `Scanner` class for console input.

Java uses `System.out` to refer to the standard output device and `System.in` to the standard input device. By default the output device is the display monitor, and the input device is

the keyboard. To perform console output, you simply use the `println` method to display a primitive value or a string to the console. Console input is not directly supported in Java, but you can use the `Scanner` class to create an object to read input from `System.in`, as follows:

```
Scanner input = new Scanner(System.in);
```

The syntax `new Scanner(System.in)` creates an object of the `Scanner` type. The syntax `Scanner input` declares that `input` is a variable whose type is `Scanner`. The whole line `Scanner input = new Scanner(System.in)` creates a `Scanner` object and assigns its reference to the variable `input`. An object may invoke its methods. To invoke a method on an object is to ask the object to perform a task. You can invoke the methods in Table 2.1 to read various types of input.

TABLE 2.1 Methods for `Scanner` Objects

Method	*Description*
`nextByte()`	reads an integer of the `byte` type.
`nextShort()`	reads an integer of the `short` type.
`nextInt()`	reads an integer of the `int` type.
`nextLong()`	reads an integer of the `long` type.
`nextFloat()`	reads a number of the `float` type.
`nextDouble()`	reads a number of the `double` type.
`next()`	reads a string that ends before a whitespace character.
`nextLine()`	reads a line of text (i.e., a string ending with the *Enter* key pressed).

For now, we will see how to read a number that includes a decimal point by invoking the `nextDouble()` method. Other methods will be covered when they are used. Listing 2.2 rewrites Listing 2.1 to prompt the user to enter a radius.

LISTING 2.2 ComputeAreaWithConsoleInput.java

```
1  import java.util.Scanner; // Scanner is in the java.util package
2
3  public class ComputeAreaWithConsoleInput {
4    public static void main(String[] args) {
5      // Create a Scanner object
6      Scanner input = new Scanner(System.in);
7
8      // Prompt the user to enter a radius
9      System.out.print("Enter a number for radius: ");
10     double radius = input.nextDouble() ;
11
12     // Compute area
13     double area = radius * radius * 3.14159;
14
15     // Display result
16     System.out.println("The area for the circle of radius " +
17       radius + " is " + area);
18   }
19 }
```

import class

create a **Scanner**

read a **double**

```
Enter a number for radius: 2.5 ↵Enter
The area for the circle of radius 2.5 is 19.6349375
```





```
Enter a number for radius: 23 Enter
The area for the circle of radius 23.0 is 1661.90111
```

The **Scanner** class is in the **java.util** package. It is imported in line 1. Line 6 creates a **Scanner** object.

The statement in line 9 displays a message to prompt the user for input.

```
System.out.print("Enter a number for radius: ");
```

print vs. **println**

The **print** method is identical to the **println** method except that **println** moves the cursor to the next line after displaying the string, but **print** does not advance the cursor to the next line when completed.

The statement in line 10 reads an input from the keyboard.

```
double radius = input.nextDouble();
```

After the user enters a number and presses the *Enter* key, the number is read and assigned to **radius**.

More details on objects will be introduced in Chapter 8, "Objects and Classes." For the time being, simply accept that this is how to obtain input from the console.

Listing 2.3 gives another example of reading input from the keyboard. The example reads three numbers and displays their average.

LISTING 2.3 ComputeAverage.java

import class

create a **Scanner**

read a **double**

```
1  import java.util.Scanner; // Scanner is in the java.util package
2
3  public class ComputeAverage {
4    public static void main(String[] args) {
5      // Create a Scanner object
6      Scanner input = new Scanner(System.in);
7
8      // Prompt the user to enter three numbers
9      System.out.print("Enter three numbers: ");
10     double number1 = input.nextDouble();
11     double number2 = input.nextDouble();
12     double number3 = input.nextDouble();
13
14     // Compute average
15     double average = (number1 + number2 + number3) / 3;
16
17     // Display result
18     System.out.println("The average of " + number1 + " " + number2
19       + " " + number3 + " is " + average);
20   }
21 }
```

enter input in one line



```
Enter three numbers: 1 2 3 Enter
The average of 1.0 2.0 3.0 is 2.0
```

enter input in multiple lines



```
Enter three numbers: 10.5 Enter
11 Enter
11.5 Enter
The average of 10.5 11.0 11.5 is 11.0
```

The code for importing the `Scanner` class (line 1) and creating a `Scanner` object (line 6) are the same as in the preceding example as well as in all new programs you will write.

Line 9 prompts the user to enter three numbers. The numbers are read in lines 10–12. You may enter three numbers separated by spaces, then press the *Enter* key, or enter each number followed by a press of the *Enter* key, as shown in the sample runs of this program.

2.4 Identifiers

As you see in Listing 2.3, `ComputeAverage`, `main`, `input`, `number1`, `number2`, `number3`, and so on are the names of things that appear in the program. Such names are called *identifiers*. All identifiers must obey the following rules:

identifier naming rules

- An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs ($).
- An identifier must start with a letter, an underscore (_), or a dollar sign ($). It cannot start with a digit.
- An identifier cannot be a reserved word. (See Appendix A, "Java Keywords," for a list of reserved words.)
- An identifier cannot be `true`, `false`, or `null`.
- An identifier can be of any length.

For example, `$2`, `ComputeArea`, `area`, `radius`, and `showMessageDialog` are legal identifiers, whereas `2A` and `d+4` are not because they do not follow the rules. The Java compiler detects illegal identifiers and reports syntax errors.



Note

Since Java is case sensitive, `area`, `Area`, and `AREA` are all different identifiers.

case sensitive



Tip

Identifiers are for naming variables, constants, methods, classes, and packages. Descriptive identifiers make programs easy to read.

descriptive names



Tip

Do not name identifiers with the `$` character. By convention, the `$` character should be used only in mechanically generated source code.

the **$** character

2.5 Variables

As you see from the programs in the preceding sections, variables are used to store values to be used later in a program. They are called variables because their values can be changed. In the program in Listing 2.2, `radius` and `area` are variables of double-precision, floating-point type. You can assign any numerical value to `radius` and `area`, and the values of `radius` and `area` can be reassigned. For example, you can write the code shown below to compute the area for different radii:

why called variables?

```
// Compute the first area
radius = 1.0;
area = radius * radius * 3.14159;
System.out.println("The area is " + area + " for radius " + radius);

// Compute the second area
radius = 2.0;
area = radius * radius * 3.14159;
System.out.println("The area is " + area + " for radius " + radius);
```

Variables are for representing data of a certain type. To use a variable, you declare it by telling the compiler its name as well as what type of data it can store. The *variable declaration* tells the compiler to allocate appropriate memory space for the variable based on its data type. The syntax for declaring a variable is

```
datatype variableName;
```

declaring variables

Here are some examples of variable declarations:

```
int count;           // Declare count to be an integer variable;
double radius;       // Declare radius to be a double variable;
double interestRate; // Declare interestRate to be a double variable;
```

The examples use the data types `int`, `double`, and `char`. Later you will be introduced to additional data types, such as `byte`, `short`, `long`, `float`, `char`, and `boolean`.

If variables are of the same type, they can be declared together, as follows:

```
datatype variable1, variable2, ..., variablen;
```

The variables are separated by commas. For example,

```
int i, j, k; // Declare i, j, and k as int variables
```



Note

naming variables

By convention, variable names are in lowercase. If a name consists of several words, concatenate all of them and capitalize the first letter of each word except the first. Examples of variables are `radius` and `interestRate`.

initializing variables

Variables often have initial values. You can declare a variable and initialize it in one step. Consider, for instance, the following code:

```
int count = 1;
```

This is equivalent to the next two statements:

```
int count;
x = 1;
```

You can also use a shorthand form to declare and initialize variables of the same type together. For example,

```
int i = 1, j = 2;
```



Tip

A variable must be declared before it can be assigned a value. A variable declared in a method must be assigned a value before it can be used.

Whenever possible, declare a variable and assign its initial value in one step. This will make the program easy to read and avoid programming errors.

2.6 Assignment Statements and Assignment Expressions

assignment statement
assignment operator

After a variable is declared, you can assign a value to it by using an *assignment statement*. In Java, the equal sign (`=`) is used as the *assignment operator*. The syntax for assignment statements is as follows:

```
variable = expression;
```

An *expression* represents a computation involving values, variables, and operators that, taking them together, evaluates to a value. For example, consider the following code: expression

```
int x = 1;                 // Assign 1 to variable x
double radius = 1.0;       // Assign 1.0 to variable radius
x = 5 * (3 / 2) + 3 * 2;   // Assign the value of the expression to x
x = y + 1;                 // Assign the addition of y and 1 to x
area = radius * radius * 3.14159; // Compute area
```

A variable can also be used in an expression. For example,

```
x = x + 1;
```

In this assignment statement, the result of `x + 1` is assigned to `x`. If `x` is `1` before the statement is executed, then it becomes `2` after the statement is executed.

To assign a value to a variable, the variable name must be on the left of the assignment operator. Thus, `1 = x` would be wrong.



Note

In mathematics, `x = 2 * x + 1` denotes an equation. However, in Java, `x = 2 * x + 1` is an assignment statement that evaluates the expression `2 * x + 1` and assigns the result to `x`.

In Java, an assignment statement is essentially an expression that evaluates to the value to be assigned to the variable on the left-hand side of the assignment operator. For this reason, an assignment statement is also known as an *assignment expression*. For example, the following statement is correct: assignment expression

```
System.out.println(x = 1);
```

which is equivalent to

```
x = 1;
System.out.println(x);
```

The following statement is also correct:

```
i = j = k = 1;
```

which is equivalent to

```
k = 1;
j = k;
i = j;
```



Note

In an assignment statement, the data type of the variable on the left must be compatible with the data type of the value on the right. For example, `int x = 1.0` would be illegal, because the data type of `x` is `int`. You cannot assign a `double` value (`1.0`) to an `int` variable without using type casting. Type casting is introduced in §2.11 "Numeric Type Conversions."

2.7 Named Constants

The value of a variable may change during the execution of a program, but a *named constant* or simply *constant* represents permanent data that never changes. In our `ComputeArea` program, π is a constant. If you use it frequently, you don't want to keep typing `3.14159`; instead, you can declare a constant for π. Here is the syntax for declaring a constant: constant

```
final datatype CONSTANTNAME = VALUE;
```

A constant must be declared and initialized in the same statement. The word `final` is a Java keyword for declaring a constant. For example, you can declare π as a constant and rewrite Listing 2.1 as follows:

```
// ComputeArea.java: Compute the area of a circle
public class ComputeArea {
  public static void main(String[] args) {
    final double PI = 3.14159; // Declare a constant

    // Assign a radius
    double radius = 20;

    // Compute area
    double area = radius * radius * PI;

    // Display results
    System.out.println("The area for the circle of radius " +
      radius + " is " + area);
  }
}
```

naming constants



Caution

By convention, constants are named in uppercase: `PI`, not `pi` or `Pi`.

benefits of constants



Note

There are three benefits of using constants: (1) you don't have to repeatedly type the same value; (2) if you have to change the constant value (e.g., from `3.14` to `3.14159` for `PI`), you need to change it only in a single location in the source code; (3) a descriptive name for a constant makes the program easy to read.

2.8 Numeric Data Types and Operations

Every data type has a range of values. The compiler allocates memory space for each variable or constant according to its data type. Java provides eight primitive data types for numeric values, characters, and Boolean values. This section introduces numeric data types.

Table 2.2 lists the six numeric data types, their ranges, and their storage sizes.

TABLE 2.2 Numeric Data Types

Name	*Range*	*Storage Size*
`byte`	-2^7 (-128) to 2^7-1 (127)	8-bit signed
`short`	-2^{15} (-32768) to $2^{15}-1$ (32767)	16-bit signed
`int`	-2^{31} (-2147483648) to $2^{31}-1$ (2147483647)	32-bit signed
`long`	-2^{63} to $2^{63}-1$ (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed
`float`	Negative range: $-3.4028235E+38$ to $-1.4E-45$ Positive range: $1.4E-45$ to $3.4028235E+38$	32-bit IEEE 754
`double`	Negative range: $-1.7976931348623157E+308$ to $-4.9E-324$ Positive range: $4.9E-324$ to $1.7976931348623157E+308$	64-bit IEEE 754



Note

IEEE 754 is a standard approved by the Institute of Electrical and Electronics Engineers for representing floating-point numbers on computers. The standard has been widely adopted. Java has adopted the 32-bit **IEEE 754** for the `float` type and the 64-bit **IEEE 754** for the `double` type. The **IEEE 754** standard also defines special values as given in Appendix E, "Special Floating-Point Values."

integer types

Java uses four types for integers: `byte`, `short`, `int`, and `long`. Choose the type that is most appropriate for your variable. For example, if you know an integer stored in a variable is within a range of byte, declare the variable as a `byte`. For simplicity and consistency, we will use `int` for integers most of the time in this book.

floating point

Java uses two types for floating-point numbers: `float` and `double`. The `double` type is twice as big as `float`. So, the `double` is known as *double precision*, `float` as *single precision*. Normally you should use the `double` type, because it is more accurate than the `float` type.



Caution

what is overflow?

When a variable is assigned a value that is too large (*in size*) to be stored, it causes *overflow*. For example, executing the following statement causes *overflow*, because the largest value that can be stored in a variable of the `int` type is `2147483647`. `2147483648` is too large.

```
int value = 2147483647 + 1; // value will actually be -2147483648
```

Likewise, executing the following statement causes *overflow*, because the smallest value that can be stored in a variable of the `int` type is `-2147483648`. `-2147483649` is too large in size to be stored in an `int` variable.

```
int value = -2147483648 - 1; // value will actually be 2147483647
```

Java does not report warnings or errors on overflow. So be careful when working with numbers close to the maximum or minimum range of a given type.

what is underflow?

When a floating-point number is too small (i.e., too close to zero) to be stored, it causes *underflow*. Java approximates it to zero. So normally you should not be concerned with underflow.

2.8.1 Numeric Operators

operators +, -, *, /, %

The operators for numeric data types include the standard arithmetic operators: addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and remainder (`%`), as shown in Table 2.3.

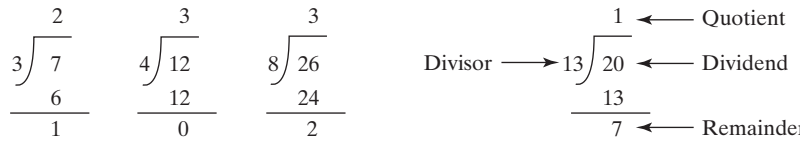
integer division

When both operands of a division are integers, the result of the division is an integer. The fractional part is truncated. For example, `5 / 2` yields `2`, not `2.5`, and `-5 / 2` yields `-2`, not `-2.5`. To perform regular mathematical division, one of the operands must be a floating-point number. For example, `5.0 / 2` yields `2.5`.

The `%` operator yields the remainder after division. The left-hand operand is the dividend and the right-hand operand the divisor. Therefore, `7 % 3` yields `1`, `12 % 4` yields `0`, `26 % 8` yields `2`, and `20 % 13` yields `7`.

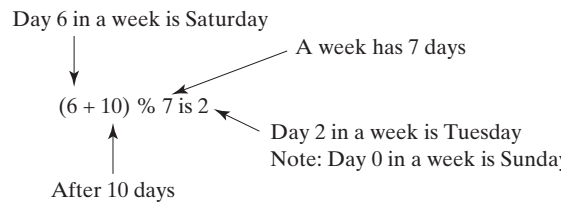
TABLE 2.3 Numeric Operators

Name	*Meaning*	*Example*	*Result*
`+`	Addition	34 + 1	35
`-`	Subtraction	34.0 − 0.1	33.9
`*`	Multiplication	300 * 30	9000
`/`	Division	1.0 / 2.0	0.5
`%`	Remainder	20 % 3	2



The `%` operator is often used for positive integers but can be used also with negative integers and floating-point values. The remainder is negative only if the dividend is negative. For example, `-7 % 3` yields `-1`, `-12 % 4` yields `0`, `-26 % -8` yields `-2`, and `20 % -13` yields `7`.

Remainder is very useful in programming. For example, an even number `% 2` is always `0` and an odd number `% 2` is always `1`. So you can use this property to determine whether a number is even or odd. If today is Saturday, it will be Saturday again in 7 days. Suppose you and your friends are going to meet in 10 days. What day is in 10 days? You can find that the day is Tuesday using the following expression:



Listing 2.4 gives a program that obtains minutes and remaining seconds from an amount of time in seconds. For example, `500` seconds contains `8` minutes and `20` seconds.

LISTING 2.4 DisplayTime.java

import **Scanner**
create a **Scanner**
read an integer
divide
remainder

```
1  import java.util.Scanner;
2
3  public class DisplayTime {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6      // Prompt the user for input
7      System.out.print("Enter an integer for seconds: ");
8      int seconds = input.nextInt() ;
9
10     int minutes = seconds / 60 ; // Find minutes in seconds
11     int remainingSeconds = seconds % 60 ; // Seconds remaining
12     System.out.println(seconds + " seconds is " + minutes +
13       " minutes and " + remainingSeconds + " seconds");
14   }
15 }
```



```
Enter an integer for seconds: 500 ↵Enter
500 seconds is 8 minutes and 20 seconds
```



line#	seconds	minutes	remainingSeconds
8	500		
10		8	
11			20

The `nextInt()` method (line 8) reads an integer for `seconds`. Line 4 obtains the minutes using `seconds / 60`. Line 5 (`seconds % 60`) obtains the remaining seconds after taking away the minutes.

The + and - operators can be both unary and binary. A *unary* operator has only one operand; a *binary* operator has two. For example, the - operator in -5 is a unary operator to negate number 5, whereas the - operator in 4 - 5 is a binary operator for subtracting 5 from 4.

unary operator
binary operator



Note

Calculations involving floating-point numbers are approximated because these numbers are not stored with complete accuracy. For example,

floating-point approximation

```
System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);
```

displays 0.5000000000000001, not 0.5, and

```
System.out.println(1.0 - 0.9);
```

displays 0.09999999999999998, not 0.1. Integers are stored precisely. Therefore, calculations with integers yield a precise integer result.

2.8.2 Numeric Literals

A *literal* is a constant value that appears directly in a program. For example, 34 and 0.305 are literals in the following statements:

literal

```
int numberOfYears = 34;
double weight = 0.305;
```

Integer Literals

An integer literal can be assigned to an integer variable as long as it can fit into the variable. A compile error will occur if the literal is too large for the variable to hold. The statement `byte b = 128`, for example, will cause a compile error, because 128 cannot be stored in a variable of the `byte` type. (Note that the range for a byte value is from −128 to 127.)

An integer literal is assumed to be of the `int` type, whose value is between -2^{31} (−2147483648) and $2^{31} - 1$ (2147483647). To denote an integer literal of the `long` type, append the letter L or l to it (e.g., 2147483648L). L is preferred because l (lowercase L) can easily be confused with 1 (the digit one). To write integer 2147483648 in a Java program, you have to write it as 2147483648L, because 2147483648 exceeds the range for the `int` value.

long literal



Note

By default, an integer literal is a decimal integer number. To denote an octal integer literal, use a leading *0* (zero), and to denote a hexadecimal integer literal, use a leading *0x* or *0X* (zero x). For example, the following code displays the decimal value 65535 for hexadecimal number FFFF.

octal and hex literals

```
System.out.println(0xFFFF);
```

Hexadecimal numbers, binary numbers, and octal numbers are introduced in Appendix F, "Number Systems."

Floating-Point Literals

Floating-point literals are written with a decimal point. By default, a floating-point literal is treated as a `double` type value. For example, 5.0 is considered a `double` value, not a `float` value. You can make a number a `float` by appending the letter `f` or `F`, and you can make a number a `double` by appending the letter `d` or `D`. For example, you can use 100.2f or 100.2F for a `float` number, and 100.2d or 100.2D for a `double` number.

suffix d or D
suffix f or F



Note

The `double` type values are more accurate than the `float` type values. For example,

double vs. **float**

```
System.out.println("1.0 / 3.0 is " + 1.0 / 3.0);
```

displays `1.0 / 3.0 is 0.3333333333333333`.

```
System.out.println("1.0F / 3.0F is " + 1.0F / 3.0F);
```

displays `1.0F / 3.0F is 0.33333334`.

Scientific Notation

Floating-point literals can also be specified in scientific notation; for example, `1.23456e+2`, the same as `1.23456e2`, is equivalent to $1.23456 \times 10^2 = 123.456$, and `1.23456e-2` is equivalent to $1.23456 \times 10^{-2} = 0.0123456$. `E` (or `e`) represents an exponent and can be in either lowercase or uppercase.

why called floating-point?



Note

The `float` and `double` types are used to represent numbers with a decimal point. Why are they called *floating-point numbers*? These numbers are stored in scientific notation. When a number such as `50.534` is converted into scientific notation, such as `5.0534e+1`, its decimal point is moved (i.e., floated) to a new position.

2.8.3 Evaluating Java Expressions

Writing a numeric expression in Java involves a straightforward translation of an arithmetic expression using Java operators. For example, the arithmetic expression

$$\frac{3 + 4x}{5} - \frac{10(y - 5)(a + b + c)}{x} + 9\left(\frac{4}{x} + \frac{9 + x}{y}\right)$$

can be translated into a Java expression as:

```
(3 + 4 * x) / 5 - 10 * (y - 5) * (a + b + c) / x +
9 * (4 / x + (9 + x) / y)
```

evaluating an expression

Though Java has its own way to evaluate an expression behind the scene, the result of a Java expression and its corresponding arithmetic expression are the same. Therefore, you can safely apply the arithmetic rule for evaluating a Java expression. Operators contained within pairs of parentheses are evaluated first. Parentheses can be nested, in which case the expression in the inner parentheses is evaluated first. Multiplication, division, and remainder operators are applied next. If an expression contains several multiplication, division, and remainder operators, they are applied from left to right. Addition and subtraction operators are applied last. If an expression contains several addition and subtraction operators, they are applied from left to right. Here is an example of how an expression is evaluated:

```
3 + 4 * 4 + 5 * (4 + 3) - 1
```
(1) inside parentheses first

```
3 + 4 * 4 + 5 * 7 - 1
```
(2) multiplication

```
3 + 16 + 5 * 7 - 1
```
(3) multiplication

```
3 + 16 + 35 - 1
```
(4) addition

```
19 + 35 - 1
```
(5) addition

```
54 - 1
```
(6) subtraction

```
53
```

Listing 2.5 gives a program that converts a Fahrenheit degree to Celsius using the formula $celsius = \left(\frac{5}{9}\right)(fahrenheit - 32)$.

LISTING 2.5 FahrenheitToCelsius.java

```
1 import java.util.Scanner;
2
3 public class FahrenheitToCelsius {
4   public static void main(String[] args) {
5     Scanner input = new Scanner(System.in);
6
7     System.out.print("Enter a degree in Fahrenheit: ");
8     double fahrenheit = input.nextDouble();
9
10    // Convert Fahrenheit to Celsius
11    double celsius = (5.0 / 9) * (fahrenheit - 32);
12    System.out.println("Fahrenheit " + fahrenheit + " is " +
13      celsius + " in Celsius");
14  }
15 }
```

divide

```
Enter a degree in Fahrenheit: 100 Enter
Fahrenheit 100.0 is 37.77777777777778 in Celsius
```





line#	fahrenheit	celsius
8	100	
11		37.77777777777778

Be careful when applying division. Division of two integers yields an integer in Java. $\frac{5}{9}$ is translated to `5.0 / 9` instead of `5 / 9` in line 11, because `5 / 9` yields `0` in Java.

integer vs. decimal division

2.9 Problem: Displaying the Current Time



Video Note
Use operators / and %

The problem is to develop a program that displays the current time in GMT (Greenwich Mean Time) in the format hour:minute:second, such as 13:19:8.

The `currentTimeMillis` method in the `System` class returns the current time in milliseconds elapsed since the time `00:00:00` on January 1, 1970 GMT, as shown in Figure 2.2. This time is known as the *Unix epoch,* because `1970` was the year when the Unix operating system was formally introduced.

currentTimeMillis

Unix epoch

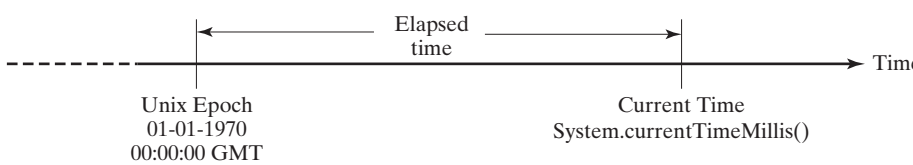


FIGURE 2.2 The `System.currentTimeMillis()` returns the number of milliseconds since the Unix epoch.

You can use this method to obtain the current time, and then compute the current second, minute, and hour as follows.

1. Obtain the total milliseconds since midnight, Jan 1, 1970, in `totalMilliseconds` by invoking `System.currentTimeMillis()` (e.g., `1203183086328` milliseconds).
2. Obtain the total seconds `totalSeconds` by dividing `totalMilliseconds` by `1000` (e.g., `1203183086328` milliseconds / `1000` = `1203183086` seconds).
3. Compute the current second from `totalSeconds % 60` (e.g., `1203183086` seconds % `60` = `26`, which is the current second).
4. Obtain the total minutes `totalMinutes` by dividing `totalSeconds` by `60` (e.g., `1203183086` seconds / `60` = `20053051` minutes).
5. Compute the current minute from `totalMinutes % 60` (e.g., `20053051` minutes % `60` = `31`, which is the current minute).
6. Obtain the total hours `totalHours` by dividing `totalMinutes` by `60` (e.g., `20053051` minutes / `60` = `334217` hours).
7. Compute the current hour from `totalHours % 24` (e.g., `334217` hours % `24` = `17`, which is the current hour).

Listing 2.6 gives the complete program.

LISTING 2.6 ShowCurrentTime.java

```
1  public class ShowCurrentTime {
2    public static void main(String[] args) {
3      // Obtain the total milliseconds since midnight, Jan 1, 1970
4      long totalMilliseconds = System.currentTimeMillis();
5
6      // Obtain the total seconds since midnight, Jan 1, 1970
7      long totalSeconds = totalMilliseconds / 1000;
8
9      // Compute the current second in the minute in the hour
10     long currentSecond = (int)(totalSeconds % 60);
11
12     // Obtain the total minutes
13     long totalMinutes = totalSeconds / 60;
14
15     // Compute the current minute in the hour
16     long currentMinute = totalMinutes % 60;
17
18     // Obtain the total hours
19     long totalHours = totalMinutes / 60;
20
21     // Compute the current hour
22     long currentHour = totalHours % 24;
23
24     // Display results
25     System.out.println("Current time is " + currentHour + ":"
26       + currentMinute + ":" + currentSecond + " GMT");
27   }
28 }
```

`totalMilliseconds` `totalSeconds` `currentSecond` `totalMinutes` `currentMinute` `totalHours` `currentHour`



```
Current time is 17:31:26 GMT
```

variables \ line#	4	7	10	13	16	19	22
totalMilliseconds	1203183086328						
totalSeconds		1203183086					
currentSecond			26				
totalMinutes				20053051			
currentMinute					31		
totalHours						334217	
currentHour							17



When `System.currentTimeMillis()` (line 4) is invoked, it returns the difference, measured in milliseconds, between the current GMT and midnight, January 1, 1970 GMT. This method returns the milliseconds as a `long` value. So, all the variables are declared as the `long` type in this program.

2.10 Shorthand Operators

Very often the current value of a variable is used, modified, and then reassigned back to the same variable. For example, the following statement adds the current value of `i` with `8` and assigns the result back to `i`:

```
i = i + 8;
```

Java allows you to combine assignment and addition operators using a shorthand operator. For example, the preceding statement can be written as:

```
i += 8;
```

The `+=` is called the *addition assignment operator*. Other shorthand operators are shown in Table 2.4.

addition assignment operator

TABLE 2.4 Shorthand Operators

Operator	*Name*	*Example*	*Equivalent*
`+=`	Addition assignment	`i += 8`	`i = i + 8`
`-=`	Subtraction assignment	`i -= 8`	`i = i - 8`
`*=`	Multiplication assignment	`i *= 8`	`i = i * 8`
`/=`	Division assignment	`i /= 8`	`i = i / 8`
`%=`	Remainder assignment	`i %= 8`	`i = i % 8`



Caution

There are no spaces in the shorthand operators. For example, `+ =` should be `+=`.



Note

Like the assignment operator (`=`), the operators (`+=`, `-=`, `*=`, `/=`, `%=`) can be used to form an assignment statement as well as an expression. For example, in the following code, `x += 2` is a statement in the first line and an expression in the second line.

```
x += 2; // Statement
System.out.println(x += 2); // Expression
```

There are two more shorthand operators for incrementing and decrementing a variable by `1`. These are handy, because that's often how much the value needs to be changed. The two operators are `++` and `--`. For example, the following code increments `i` by `1` and decrements `j` by `1`.

```
int i = 3, j = 3;
i++; // i becomes 4
j--; // j becomes 2
```

The `++` and `--` operators can be used in prefix or suffix mode, as shown in Table 2.5.

TABLE 2.5 Increment and Decrement Operators

Operator	*Name*	*Description*	*Example (assume i = 1)*
`++var`	preincrement	Increment `var` by `1` and use the new `var` value	`int j = ++i; // j is 2, // i is 2`
`var++`	postincrement	Increment `var` by `1`, but use the original `var` value	`int j = i++; // j is 1, // i is 2`
`--var`	predecrement	Decrement `var` by `1` and use the new `var` value	`int j = --i; // j is 0, // i is 0`
`var--`	postdecrement	Decrement `var` by `1` and use the original `var` value	`int j = ++i; // j is 1, // i is 0`

preincrement, predecrement
postincrement, postdecrement

If the operator is *before* (prefixed to) the variable, the variable is incremented or decremented by `1`, then the *new* value of the variable is returned. If the operator is *after* (suffixed to) the variable, then the variable is incremented or decremented by `1`, but the original *old* value of the variable is returned. Therefore, the prefixes `++x` and `--x` are referred to, respectively, as the *preincrement operator* and the *predecrement operator*; and the suffixes `x++` and `x--` are referred to, respectively, as the *postincrement operator* and the *postdecrement operator*. The prefix form of `++` (or `--`) and the suffix form of `++` (or `--`) are the same if they are used in isolation, but they cause different effects when used in an expression. The following code illustrates this:

```
int i = 10;
int newNum = 10 * i++;
```

Same effect as

```
int newNum = 10 * i;
i = i + 1;
```

In this case, `i` is incremented by `1`, then the *old* value of `i` is returned and used in the multiplication. So `newNum` becomes `100`. If `i++` is replaced by `++i` as follows,

```
int i = 10;
int newNum = 10 * (++i);
```

Same effect as

```
i = i + 1;
int newNum = 10 * i;
```

`i` is incremented by `1`, and the new value of `i` is returned and used in the multiplication. Thus `newNum` becomes `110`.

Here is another example:

```
double x = 1.0;
double y = 5.0;
double z = x-- + (++y);
```

After all three lines are executed, `y` becomes `6.0`, `z` becomes `7.0`, and `x` becomes `0.0`.

The increment operator `++` and the decrement operator `--` can be applied to all integer and floating-point types. These operators are often used in loop statements. A *loop statement* is a construct that controls how many times an operation or a sequence of operations is performed in succession. This construct, and the topic of loop statements, are introduced in Chapter 4, "Loops."



Tip

Using increment and decrement operators makes expressions short, but it also makes them complex and difficult to read. Avoid using these operators in expressions that modify multiple variables or the same variable multiple times, such as this one: `int k = ++i + i`.

2.11 Numeric Type Conversions

Can you perform binary operations with two operands of different types? Yes. If an integer and a floating-point number are involved in a binary operation, Java automatically converts the integer to a floating-point value. So, `3 * 4.5` is same as `3.0 * 4.5`.

You can always assign a value to a numeric variable whose type supports a larger range of values; thus, for instance, you can assign a `long` value to a `float` variable. You cannot, however, assign a value to a variable of a type with smaller range unless you use *type casting*. Casting is an operation that converts a value of one data type into a value of another data type. Casting a variable of a type with a small range to a variable of a type with a larger range is known as *widening a type*. Casting a variable of a type with a large range to a variable of a type with a smaller range is known as *narrowing a type*. Widening a type can be performed automatically without explicit casting. Narrowing a type must be performed explicitly.

widening a type

narrowing a type

The syntax is the target type in parentheses, followed by the variable's name or the value to be cast. For example, the following statement

type casting

```
System.out.println((int)1.7);
```

displays `1`. When a `double` value is cast into an `int` value, the fractional part is truncated.

The following statement

```
System.out.println((double)1 / 2);
```

displays `0.5`, because `1` is cast to `1.0` first, then `1.0` is divided by `2`. However, the statement

```
System.out.println(1 / 2);
```

displays `0`, because `1` and `2` are both integers and the resulting value should also be an integer.



Caution

Casting is necessary if you are assigning a value to a variable of a smaller type range, such as assigning a `double` value to an `int` variable. A compile error will occur if casting is not used in situations of this kind. Be careful when using casting. Loss of information might lead to inaccurate results.

possible loss of precision



Note

Casting does not change the variable being cast. For example, `d` is not changed after casting in the following code:

```
double d = 4.5;
int i = (int)d;  // i becomes 4, but d is not changed, still 4.5
```



Note

To assign a variable of the `int` type to a variable of the `short` or `byte` type, explicit casting must be used. For example, the following statements have a compile error:

```
int i = 1;
byte b = i; // Error because explicit casting is required
```

However, so long as the integer literal is within the permissible range of the target variable, explicit casting is not needed to assign an integer literal to a variable of the `short` or `byte` type. Please refer to §2.8.2, "Numeric Literals."

Listing 2.7 gives a program that displays the sales tax with two digits after the decimal point.

LISTING 2.7 SalesTax.java

```
1 import java.util.Scanner;
2
3 public class SalesTax {
4   public static void main(String[] args) {
5     Scanner input = new Scanner(System.in);
6
7     System.out.print("Enter purchase amount: ");
8     double purchaseAmount = input.nextDouble();
9
10    double tax = purchaseAmount * 0.06;
11    System.out.println("Sales tax is " + (int)(tax * 100) / 100.0);
12  }
13 }
```

casting



```
Enter purchase amount: 197.55 Enter
Sales tax is 11.85
```



line#	purchaseAmount	tax	output
8	197.55		
10		11.853	
11			11.85

formatting numbers

Variable `purchaseAmount` is `197.55` (line 8). The sales tax is `6%` of the purchase, so the `tax` is evaluated as `11.853` (line 10). Note that

```
tax * 100 is 1185.3
(int)(tax * 100) is 1185
(int)(tax * 100) / 100.0 is 11.85
```

So, the statement in line 11 displays the tax `11.85` with two digits after the decimal point.

2.12 Problem: Computing Loan Payments



Video Note
Program computations

The problem is to write a program that computes loan payments. The loan can be a car loan, a student loan, or a home mortgage loan. The program lets the user enter the interest rate, number of years, and loan amount, and displays the monthly and total payments.

The formula to compute the monthly payment is as follows:

$$monthlyPayment = \frac{loanAmount \times monthlyInterestRate}{1 - \frac{1}{(1 + monthlyInterestRate)^{numberOfYears \times 12}}}$$

You don't have to know how this formula is derived. Nonetheless, given the monthly interest rate, number of years, and loan amount, you can use it to compute the monthly payment.

In the formula, you have to compute $(1 + monthlyInterestRate)^{numberOfYears \times 12}$. The `pow(a, b)` method in the `Math` class can be used to compute a^b. The `Math` class, which comes with the Java API, is available to all Java programs. For example,

`pow(a, b)` method

```
System.out.println(Math.pow(2, 3)); // Display 8
System.out.println(Math.pow(4, 0.5)); // Display 4
```

$(1 + monthlyInterestRate)^{numberOfYears \times 12}$ can be computed using `Math.pow(1 + monthlyInterestRate, numberOfYears * 12)`.

Here are the steps in developing the program:

1. Prompt the user to enter the annual interest rate, number of years, and loan amount.
2. Obtain the monthly interest rate from the annual interest rate.
3. Compute the monthly payment using the preceding formula.
4. Compute the total payment, which is the monthly payment multiplied by `12` and multiplied by the number of years.
5. Display the monthly payment and total payment.

Listing 2.8 gives the complete program.

LISTING 2.8 ComputeLoan.java

```
1  import java.util.Scanner;                                          import class
2
3  public class ComputeLoan {
4    public static void main(String[] args) {
5      // Create a Scanner
6      Scanner input = new Scanner(System.in);                        create a Scanner
7
8      // Enter yearly interest rate
9      System.out.print("Enter yearly interest rate, for example 8.25: ");
10     double annualInterestRate = input.nextDouble();                enter interest rate
11
12     // Obtain monthly interest rate
13     double monthlyInterestRate = annualInterestRate / 1200;
14
15     // Enter number of years
16     System.out.print(
17       "Enter number of years as an integer, for example 5: ");
18     int numberOfYears = input.nextInt();                           enter years
19
20     // Enter loan amount
21     System.out.print("Enter loan amount, for example 120000.95: ");
```

enter loan amount

```
22      double loanAmount = input.nextDouble();
23
24      // Calculate payment
25      double monthlyPayment  = loanAmount * monthlyInterestRate / (1
26        - 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
27      double totalPayment = monthlyPayment * numberOfYears * 12;
28
29      // Display results
30     System.out.println("The monthly payment is " +
31       (int)(monthlyPayment * 100) / 100.0);
32     System.out.println("The total payment is " +
33       (int)(totalPayment * 100) / 100.0);
34   }
35 }
```

monthlyPayment

totalPayment

casting

casting



```
Enter yearly interest rate, for example 8.25: 5.75 Enter
Enter number of years as an integer, for example 5: 15 Enter
Enter loan amount, for example 120000.95: 250000 Enter
The monthly payment is 2076.02
The total payment is 373684.53
```



variables \ line#	10	13	18	22	25	27
annualInterestRate	5.75					
monthlyInterestRate		0.0047916666666				
numberOfYears			15			
loanAmount				250000		
monthlyPayment					2076.0252175	
totalPayment						373684.539

Line 10 reads the yearly interest rate, which is converted into monthly interest rate in line 13. If you entered an input other than a numeric value, a runtime error would occur.

Choose the most appropriate data type for the variable. For example, **`numberOfYears`** is best declared as an **`int`** (line 18), although it could be declared as a **`long`**, **`float`**, or **`double`**. Note that **`byte`** might be the most appropriate for **`numberOfYears`**. For simplicity, however, the examples in this book will use **`int`** for integer and **`double`** for floating-point values.

The formula for computing the monthly payment is translated into Java code in lines 25–27.

Casting is used in lines 31 and 33 to obtain a new **`monthlyPayment`** and **`totalPayment`** with two digits after the decimal point.

The program uses the **`Scanner`** class, imported in line 1. The program also uses the **`Math`** class; why isn't it imported? The **`Math`** class is in the **`java.lang`** package. All classes in the **`java.lang`** package are implicitly imported. So, there is no need to explicitly import the **`Math`** class.

`java.lang` package

2.13 Character Data Type and Operations

`char` type

The character data type, **`char`**, is used to represent a single character. A character literal is enclosed in single quotation marks. Consider the following code:

```
char letter = 'A';
char numChar = '4';
```

The first statement assigns character `A` to the `char` variable `letter`. The second statement assigns digit character `4` to the `char` variable `numChar`.



Caution

A string literal must be enclosed in quotation marks. A character literal is a single character enclosed in single quotation marks. So `"A"` is a string, and `'A'` is a character.

char literal

2.13.1 Unicode and ASCII code

Computers use binary numbers internally. A character is stored in a computer as a sequence of 0s and 1s. Mapping a character to its binary representation is called *encoding*. There are different ways to encode a character. How characters are encoded is defined by an *encoding scheme*.

character encoding

Java supports *Unicode*, an encoding scheme established by the Unicode Consortium to support the interchange, processing, and display of written texts in the world's diverse languages. Unicode was originally designed as a 16-bit character encoding. The primitive data type `char` was intended to take advantage of this design by providing a simple data type that could hold any character. However, it turned out that the `65,536` characters possible in a 16-bit encoding are not sufficient to represent all the characters in the world. The Unicode standard therefore has been extended to allow up to `1,112,064` characters. Those characters that go beyond the original 16-bit limit are called *supplementary characters*. Java supports supplementary characters. The processing and representing of supplementary characters are beyond the scope of this book. For simplicity, this book considers only the original 16-bit Unicode characters. These characters can be stored in a `char` type variable.

Unicode

original Unicode

supplementary Unicode

A 16-bit Unicode takes two bytes, preceded by `\u`, expressed in four hexadecimal digits that run from `'\u0000'` to `'\uFFFF'`. For example, the word "welcome" is translated into Chinese using two characters, 欢迎 . The Unicodes of these two characters are "`\u6B22\u8FCE`".

Listing 2.9 gives a program that displays two Chinese characters and three Greek letters.

LISTING 2.9 DisplayUnicode.java

```
1 import javax.swing.JOptionPane;
2
3 public class DisplayUnicode {
4   public static void main(String[] args) {
5     JOptionPane.showMessageDialog(null,
6       "\u6B22\u8FCE \u03b1 \u03b2 \u03b3",
7       "\u6B22\u8FCE Welcome",
8       JOptionPane.INFORMATION_MESSAGE);
9   }
10 }
```



If no Chinese font is installed on your system, you will not be able to see the Chinese characters. The Unicodes for the Greek letters α β γ are `\u03b1 \u03b2 \u03b3`.

Most computers use *ASCII* (*American Standard Code for Information Interchange*), a 7-bit encoding scheme for representing all uppercase and lowercase letters, digits, punctuation marks, and control characters. Unicode includes ASCII code, with `'\u0000'` to `'\u007F'` corresponding to the 128 ASCII characters. (See Appendix B, "The ASCII Character Set," for a list of ASCII characters and their decimal and hexadecimal codes.) You can use ASCII characters such as `'X'`, `'1'`, and `'$'` in a Java program as well as Unicodes. Thus, for example, the following statements are equivalent:

ASCII

```
char letter = 'A';
char letter = '\u0041'; // Character A's Unicode is 0041
```

Both statements assign character `A` to `char` variable `letter`.

char increment and decrement



Note

The increment and decrement operators can also be used on **char** variables to get the next or preceding Unicode character. For example, the following statements display character **b**.

```
char ch = 'a';
System.out.println(++ch);
```

2.13.2 Escape Sequences for Special Characters

Suppose you want to print a message with quotation marks in the output. Can you write a statement like this?

```
System.out.println("He said "Java is fun"");
```

No, this statement has a syntax error. The compiler thinks the second quotation character is the end of the string and does not know what to do with the rest of characters.

backslash

To overcome this problem, Java defines escape sequences to represent special characters, as shown in Table 2.6. An escape sequence begins with the backslash character (\) followed by a character that has a special meaning to the compiler.

TABLE 2.6 Java Escape Sequences

Character Escape Sequence	*Name*	*Unicode Code*
\b	Backspace	\u0008
\t	Tab	\u0009
\n	Linefeed	\u000A
\f	Formfeed	\u000C
\r	Carriage Return	\u000D
\\	Backslash	\u005C
\'	Single Quote	\u0027
\"	Double Quote	\u0022

So, now you can print the quoted message using the following statement:

```
System.out.println("He said \"Java is fun\"");
```

The output is

```
He said "Java is fun"
```

2.13.3 Casting between char and Numeric Types

A **char** can be cast into any numeric type, and vice versa. When an integer is cast into a **char**, only its lower 16 bits of data are used; the other part is ignored. For example:

```
char ch = (char)0XAB0041; // the lower 16 bits hex code 0041 is
                          // assigned to ch
System.out.println(ch);   // ch is character A
```

When a floating-point value is cast into a **char**, the floating-point value is first cast into an **int**, which is then cast into a **char**.

```
char ch = (char)65.25;    // decimal 65 is assigned to ch
System.out.println(ch);   // ch is character A
```

When a `char` is cast into a numeric type, the character's Unicode is cast into the specified numeric type.

```
int i = (int)'A'; // the Unicode of character A is assigned to i
System.out.println(i);  // i is 65
```

Implicit casting can be used if the result of a casting fits into the target variable. Otherwise, explicit casting must be used. For example, since the Unicode of `'a'` is `97`, which is within the range of a byte, these implicit castings are fine:

```
byte b = 'a';
int i = 'a';
```

But the following casting is incorrect, because the Unicode `\uFFF4` cannot fit into a byte:

```
byte b = '\uFFF4';
```

To force assignment, use explicit casting, as follows:

```
byte b = (byte)'\uFFF4';
```

Any positive integer between `0` and `FFFF` in hexadecimal can be cast into a character implicitly. Any number not in this range must be cast into a `char` explicitly.



Note

All numeric operators can be applied to `char` operands. A `char` operand is automatically cast into a number if the other operand is a number or a character. If the other operand is a string, the character is concatenated with the string. For example, the following statements

numeric operators on characters

```
int i = '2' + '3'; // (int)'2' is 50 and (int)'3' is 51
System.out.println("i is " + i); // i is 101

int j = 2 + 'a'; // (int)'a' is 97
System.out.println("j is " + j); // j is 99
System.out.println(j + " is the Unicode for character "
  + (char)j);

System.out.println("Chapter " + '2');
```

display

```
i is 101
j is 99
99 is the Unicode for character c
Chapter 2
```



Note

The Unicodes for lowercase letters are consecutive integers starting from the Unicode for `'a'`, then for `'b'`, `'c'`, ..., and `'z'`. The same is true for the uppercase letters. Furthermore, the Unicode for `'a'` is greater than the Unicode for `'A'`. So `'a' - 'A'` is the same as `'b' - 'B'`. For a lowercase letter *ch*, its corresponding uppercase letter is `(char)('A' + (ch - 'a'))`.

2.14 Problem: Counting Monetary Units

Suppose you want to develop a program that classifies a given amount of money into smaller monetary units. The program lets the user enter an amount as a `double` value representing a total in dollars and cents, and outputs a report listing the monetary equivalent in dollars, quarters, dimes, nickels, and pennies, as shown in the sample run.

Your program should report the maximum number of dollars, then the maximum number of quarters, and so on, in this order.

Here are the steps in developing the program:

1. Prompt the user to enter the amount as a decimal number, such as `11.56`.
2. Convert the amount (e.g., `11.56`) into cents (`1156`).
3. Divide the cents by `100` to find the number of dollars. Obtain the remaining cents using the cents remainder `100`.
4. Divide the remaining cents by `25` to find the number of quarters. Obtain the remaining cents using the remaining cents remainder `25`.
5. Divide the remaining cents by `10` to find the number of dimes. Obtain the remaining cents using the remaining cents remainder `10`.
6. Divide the remaining cents by `5` to find the number of nickels. Obtain the remaining cents using the remaining cents remainder `5`.
7. The remaining cents are the pennies.
8. Display the result.

The complete program is given in Listing 2.10.

LISTING 2.10 ComputeChange.java

import class

```
1 import java.util.Scanner;
2
3 public class ComputeChange {
4   public static void main(String[] args) {
5     // Create a Scanner
6     Scanner input = new Scanner(System.in);
7
8     // Receive the amount
9     System.out.print(
10      "Enter an amount in double, for example 11.56: ");
```

enter input

```
11    double amount = input.nextDouble();
12
13    int remainingAmount = (int)(amount * 100);
14
15    // Find the number of one dollars
```

dollars

```
16    int numberOfOneDollars = remainingAmount / 100;
17    remainingAmount = remainingAmount % 100;
18
19    // Find the number of quarters in the remaining amount
```

quarters

```
20    int numberOfQuarters = remainingAmount / 25;
21    remainingAmount = remainingAmount % 25;
22
23    // Find the number of dimes in the remaining amount
```

dimes

```
24    int numberOfDimes = remainingAmount / 10;
25    remainingAmount = remainingAmount % 10;
26
27    // Find the number of nickels in the remaining amount
```

nickels

```
28    int numberOfNickels = remainingAmount / 5;
29    remainingAmount = remainingAmount % 5;
30
31    // Find the number of pennies in the remaining amount
```

pennies

```
32    int numberOfPennies = remainingAmount;
33
34    // Display results
```

prepare output

```
35    System.out.println("Your amount " + amount + " consists of \n" +
```

```
36        "\t" + numberOfOneDollars + " dollars\n" +
37        "\t" + numberOfQuarters + " quarters\n" +
38        "\t" + numberOfDimes + " dimes\n" +
39        "\t" + numberOfNickels + " nickels\n" +
40        "\t" + numberOfPennies + " pennies");
41    }
42 }
```

```
Enter an amount in double, for example 11.56: 11.56  Enter
Your amount 11.56 consists of
        11 dollars
        2 quarters
        0 dimes
        1 nickels
        1 pennies
```





variables \ line#	11	13	16	17	20	21	24	25	28	29	32
Amount	11.56										
remainingAmount		1156		56		6		6		1	
numberOfOneDollars			11								
numberOfQuarters					2						
numberOfDimes							0				
numberOfNickles									1		
numberOfPennies											1

The variable **amount** stores the amount entered from the console (line 11). This variable is not changed, because the amount has to be used at the end of the program to display the results. The program introduces the variable **remainingAmount** (line 13) to store the changing **remainingAmount**.

The variable **amount** is a **double** decimal representing dollars and cents. It is converted to an **int** variable **remainingAmount**, which represents all the cents. For instance, if **amount** is **11.56**, then the initial **remainingAmount** is **1156**. The division operator yields the integer part of the division. So **1156 / 100** is **11**. The remainder operator obtains the remainder of the division. So **1156 % 100** is **56**.

The program extracts the maximum number of singles from the total amount and obtains the remaining amount in the variable **remainingAmount** (lines 16–17). It then extracts the maximum number of quarters from **remainingAmount** and obtains a new **remainingAmount** (lines 20–21). Continuing the same process, the program finds the maximum number of dimes, nickels, and pennies in the remaining amount.

loss of precision

One serious problem with this example is the possible loss of precision when casting a **double** amount to an **int remainingAmount**. This could lead to an inaccurate result. If you try to enter the amount **10.03**, **10.03 * 100** becomes **1002.9999999999999**. You will find that the program displays **10** dollars and **2** pennies. To fix the problem, enter the amount as an integer value representing cents (see Exercise 2.9).

As shown in the sample run, 0 dimes, 1 nickels, and 1 pennies are displayed in the result. It would be better not to display 0 dimes, and to display 1 nickel and 1 penny using the singular forms of the words. You will learn how to use selection statements to modify this program in the next chapter (see Exercise 3.7).

2.15 The `String` Type

The `char` type represents only one character. To represent a string of characters, use the data type called `String`. For example, the following code declares the message to be a string with value "Welcome to Java".

```
String message = "Welcome to Java";
```

`String` is actually a predefined class in the Java library just like the classes `System`, `JOptionPane`, and `Scanner`. The `String` type is not a primitive type. It is known as a *reference type*. Any Java class can be used as a reference type for a variable. Reference data types will be thoroughly discussed in Chapter 8, "Objects and Classes." For the time being, you need to know only how to declare a `String` variable, how to assign a string to the variable, and how to concatenate strings.

concatenating strings and numbers

As first shown in Listing 2.1, two strings can be concatenated. The plus sign (+) is the concatenation operator if one of the operands is a string. If one of the operands is a nonstring (e.g., a number), the nonstring value is converted into a string and concatenated with the other string. Here are some examples:

```
// Three strings are concatenated
String message = "Welcome " + "to " + "Java";

// String Chapter is concatenated with number 2
String s = "Chapter" + 2; // s becomes Chapter2

// String Supplement is concatenated with character B
String s1 = "Supplement" + 'B'; // s1 becomes SupplementB
```

If neither of the operands is a string, the plus sign (+) is the addition operator that adds two numbers.

The shorthand `+=` operator can also be used for string concatenation. For example, the following code appends the string "and Java is fun" with the string "Welcome to Java" in `message`.

```
message += " and Java is fun";
```

So the new `message` is "Welcome to Java and Java is fun".

Suppose that `i = 1` and `j = 2`, what is the output of the following statement?

```
System.out.println("i + j is " + i + j);
```

The output is "i + j is 12" because `"i + j is "` is concatenated with the value of `i` first. To force `i + j` to be executed first, enclose `i + j` in the parentheses, as follows:

```
System.out.println("i + j is " + (i + j));
```

reading strings

To read a string from the console, invoke the `next()` method on a `Scanner` object. For example, the following code reads three strings from the keyboard:

```
Scanner input = new Scanner(System.in);
System.out.println("Enter three strings: ");
String s1 = input.next();
```

```
String s2 = input.next();
String s3 = input.next();
System.out.println("s1 is " + s1);
System.out.println("s2 is " + s2);
System.out.println("s3 is " + s3);
```

```
Enter a string: Welcome to Java Enter
s1 is Welcome
s2 is to
s3 is Java
```



The `next()` method reads a string that ends with a whitespace character (i.e., `' '`, `'\t'`, `'\f'`, `'\r'`, or `'\n'`).

You can use the `nextLine()` method to read an entire line of text. The `nextLine()` method reads a string that ends with the *Enter* key pressed. For example, the following statements read a line of text.

```
Scanner input = new Scanner(System.in);
System.out.println("Enter a string: ");
String s = input.nextLine();
System.out.println("The string entered is " + s);
```

```
Enter a string: Welcome to Java Enter
The string entered is "Welcome to Java"
```





Important Caution

To *avoid input errors*, do not use `nextLine()` after `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`, and `next()`. The reasons will be explained in §9.7.3, "How Does `Scanner` Work?"

avoiding input errors

2.16 Programming Style and Documentation

Programming style deals with what programs look like. A program can compile and run properly even if written on only one line, but writing it all on one line would be bad programming style because it would be hard to read. *Documentation* is the body of explanatory remarks and comments pertaining to a program. Programming style and documentation are as important as coding. Good programming style and appropriate documentation reduce the chance of errors and make programs easy to read. So far you have learned some good programming styles. This section summarizes them and gives several guidelines. More detailed guidelines can be found in Supplement I.D, "Java Coding Style Guidelines," on the Companion Website.

programming style

documentation

2.16.1 Appropriate Comments and Comment Styles

Include a summary at the beginning of the program to explain what the program does, its key features, and any unique techniques it uses. In a long program, you should also include comments that introduce each major step and explain anything that is difficult to read. It is important to make comments concise so that they do not crowd the program or make it difficult to read.

In addition to line comment `//` and block comment `/*`, Java supports comments of a special type, referred to as *javadoc comments*. javadoc comments begin with `/**` and end with `*/`. They can be extracted into an HTML file using JDK's `javadoc` command. For more information, see java.sun.com/j2se/javadoc.

javadoc comment

Use javadoc comments (`/** ... */`) for commenting on an entire class or an entire method. These comments must precede the class or the method header in order to be extracted in a javadoc HTML file. For commenting on steps inside a method, use line comments (`//`).

2.16.2 Naming Conventions

Make sure that you choose descriptive names with straightforward meanings for the variables, constants, classes, and methods in your program. Names are case sensitive. Listed below are the conventions for naming variables, methods, and classes.

naming variables and methods

- Use lowercase for variables and methods. If a name consists of several words, concatenate them into one, making the first word lowercase and capitalizing the first letter of each subsequent word—for example, the variables `radius` and `area` and the method `showInputDialog`.

naming classes

- Capitalize the first letter of each word in a class name—for example, the class names `ComputeArea`, `Math`, and `JOptionPane`.

naming constants

- Capitalize every letter in a constant, and use underscores between words—for example, the constants `PI` and `MAX_VALUE`.

It is important to follow the naming conventions to make programs easy to read.

naming classes



Caution

Do not choose class names that are already used in the Java library. For example, since the `Math` class is defined in Java, you should not name your class `Math`.

using full descriptive names



Tip

Avoid using abbreviations for identifiers. Using complete words is more descriptive. For example, `numberOfStudents` is better than `numStuds`, `numOfStuds`, or `numOfStudents`.

2.16.3 Proper Indentation and Spacing

indent code

A consistent indentation style makes programs clear and easy to read, debug, and maintain. *Indentation* is used to illustrate the structural relationships between a program's components or statements. Java can read the program even if all of the statements are in a straight line, but humans find it easier to read and maintain code that is aligned properly. Indent each subcomponent or statement at least *two* spaces more than the construct within which it is nested.

A single space should be added on both sides of a binary operator, as shown in the following statement:

```
int i= 3+4 * 4;   <-- Bad style
int i = 3 + 4 * 4;   <-- Good style
```

A single space line should be used to separate segments of the code to make the program easier to read.

2.16.4 Block Styles

A block is a group of statements surrounded by braces. There are two popular styles, *next-line* style and *end-of-line* style, as shown below.

```
public class Test
{
  public static void main(String[] args)
  {
    System.out.println("Block Styles");
  }
}
```

Next-line style

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Block Styles");
  }
}
```

End-of-line style

The next-line style aligns braces vertically and makes programs easy to read, whereas the end-of-line style saves space and may help avoid some subtle programming errors. Both are acceptable block styles. The choice depends on personal or organizational preference. You should use a block style consistently. Mixing styles is not recommended. This book uses the *end-of-line* style to be consistent with the Java API source code.

2.17 Programming Errors

Programming errors are unavoidable, even for experienced programmers. Errors can be categorized into three types: syntax errors, runtime errors, and logic errors.

2.17.1 Syntax Errors

Errors that occur during compilation are called *syntax errors* or *compile errors*. Syntax errors result from errors in code construction, such as mistyping a keyword, omitting some necessary punctuation, or using an opening brace without a corresponding closing brace. These errors are usually easy to detect, because the compiler tells you where they are and what caused them. For example, the following program has a syntax error, as shown in Figure 2.3.

syntax errors

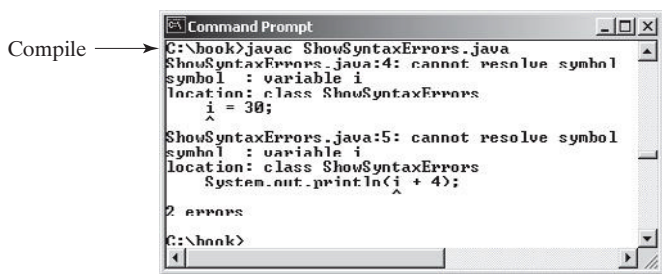


FIGURE 2.3 The compiler reports syntax errors.

```
1 // ShowSyntaxErrors.java: The program contains syntax errors
2 public class ShowSyntaxErrors {
3   public static void main(String[] args) {
4     i = 30;
5     System.out.println(i + 4);
6   }
7 }
```

syntax error

Two errors are detected. Both are the result of not declaring variable `i`. Since a single error will often display many lines of compile errors, it is a good practice to start debugging from the top line and work downward. Fixing errors that occur earlier in the program may also fix additional errors that occur later.

2.17.2 Runtime Errors

runtime errors

Runtime errors are errors that cause a program to terminate abnormally. They occur while a program is running if the environment detects an operation that is impossible to carry out. Input errors typically cause runtime errors.

An *input error* occurs when the user enters an unexpected input value that the program cannot handle. For instance, if the program expects to read in a number, but instead the user enters a string, this causes data-type errors to occur in the program. To prevent input errors, the program should prompt the user to enter values of the correct type. It may display a message such as "Please enter an integer" before reading an integer from the keyboard.

Another common source of runtime errors is division by zero. This happens when the divisor is zero for integer divisions. For instance, the following program would cause a runtime error, as shown in Figure 2.4.



FIGURE 2.4 The runtime error causes the program to terminate abnormally.

runtime error

```
1 // ShowRuntimeErrors.java: Program contains runtime errors
2 public class ShowRuntimeErrors {
3   public static void main(String[] args) {
4     int i = 1 / 0;
5   }
6 }
```

2.17.3 Logic Errors

Logic errors occur when a program does not perform the way it was intended to. Errors of this kind occur for many different reasons. For example, suppose you wrote the following program to add `number1` to `number2`.

```
// ShowLogicErrors.java: The program contains a logic error
public class ShowLogicErrors {
  public static void main(String[] args) {
    // Add number1 to number2
    int number1 = 3;
    int number2 = 3;
    number2 += number1 + number2;
    System.out.println("number2 is " + number2);
  }
}
```

The program does not have syntax errors or runtime errors, but it does not print the correct result for `number2`. See if you can find the error.

2.17.4 Debugging

In general, syntax errors are easy to find and easy to correct, because the compiler gives indications as to where the errors came from and why they are wrong. Runtime errors are not difficult to find, either, since the reasons and locations of the errors are displayed on the console when the program aborts. Finding logic errors, on the other hand, can be very challenging.

Logic errors are called *bugs*. The process of finding and correcting errors is called *debugging*. A common approach is to use a combination of methods to narrow down to the part of the program where the bug is located. You can *hand-trace* the program (i.e., catch errors by reading the program), or you can insert print statements in order to show the values of the variables or the execution flow of the program. This approach might work for a short, simple program. But for a large, complex program, the most effective approach is to use a debugger utility.

bugs
debugging
hand traces



Pedagogical NOTE

An IDE not only helps debug errors but also is an effective pedagogical tool. Supplement II shows you how to use a debugger to trace programs and how debugging can help you to learn Java effectively.

learning tool

2.18 (GUI) Getting Input from Input Dialogs

You can obtain input from the console. Alternatively, you may obtain input from an input dialog box by invoking the `JOptionPane.showInputDialog` method, as shown in Figure 2.5.

`JOptionPane` class

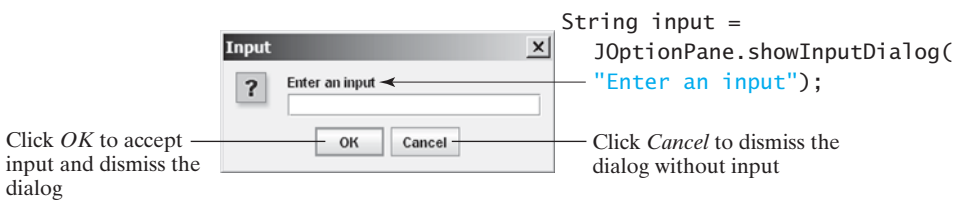


FIGURE 2.5 The input dialog box enables the user to enter a string.

When this method is executed, a dialog is displayed to enable you to enter an input value. After entering a string, click *OK* to accept the input and dismiss the dialog box. The input is returned from the method as a string.

There are several ways to use the `showInputDialog` method. For the time being, you need to know only two ways to invoke it.

`showInputDialog` method

One is to use a statement like this one:

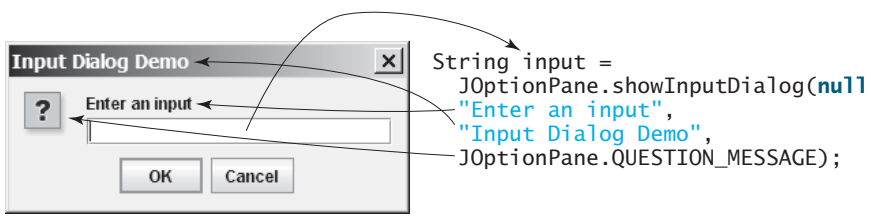
```
JOptionPane.showInputDialog(x);
```

where `x` is a string for the prompting message.

The other is to use a statement such as the following:

```
String string = JOptionPane.showInputDialog(null, x,
  y, JOptionPane.QUESTION_MESSAGE);
```

where `x` is a string for the prompting message and `y` is a string for the title of the input dialog box, as shown in the example below.



2.18.1 Converting Strings to Numbers

The input returned from the input dialog box is a string. If you enter a numeric value such as `123`, it returns `"123"`. You have to convert a string into a number to obtain the input as a number.

`Integer.parseInt` method

To convert a string into an `int` value, use the `parseInt` method in the `Integer` class, as follows:

```
int intValue = Integer.parseInt(intString);
```

where `intString` is a numeric string such as `"123"`.

`Double.parseDouble` method

To convert a string into a `double` value, use the `parseDouble` method in the `Double` class, as follows:

```
double doubleValue = Double.parseDouble(doubleString);
```

where `doubleString` is a numeric string such as `"123.45"`.

The `Integer` and `Double` classes are both included in the `java.lang` package, and thus they are automatically imported.

2.18.2 Using Input Dialog Boxes

Listing 2.8, ComputeLoan.java, reads input from the console. Alternatively, you can use input dialog boxes.

Listing 2.11 gives the complete program. Figure 2.6 shows a sample run of the program.

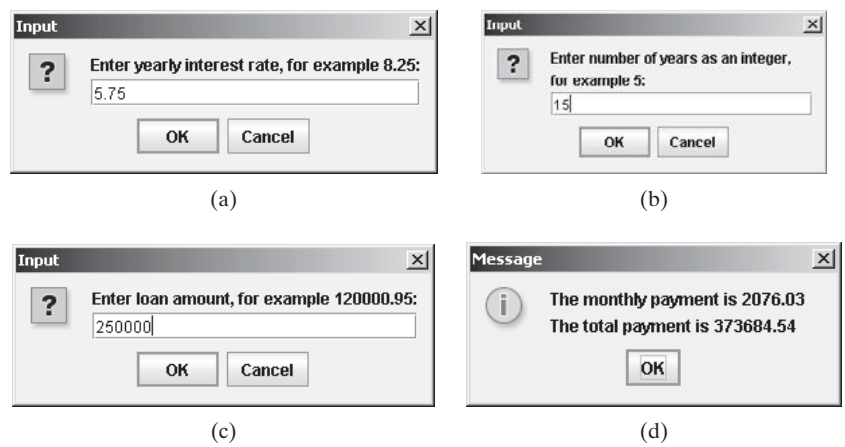


FIGURE 2.6 The program accepts the annual interest rate (a), number of years (b), and loan amount (c), then displays the monthly payment and total payment (d).

LISTING 2.11 ComputeLoanUsingInputDialog.java

```
1 import javax.swing.JOptionPane;
2
3 public class ComputeLoanUsingInputDialog {
4   public static void main(String[] args) {
5     // Enter yearly interest rate
6     String annualInterestRateString = JOptionPane.showInputDialog(
7       "Enter yearly interest rate, for example 8.25:");
8
9     // Convert string to double
10    double annualInterestRate =
11      Double.parseDouble(annualInterestRateString);
12
```

enter interest rate

convert string to double

```
13      // Obtain monthly interest rate
14      double monthlyInterestRate = annualInterestRate / 1200;
15
16      // Enter number of years
17      String numberOfYearsString = JOptionPane.showInputDialog(
18        "Enter number of years as an integer, \nfor example 5:");
19
20      // Convert string to int
21      int numberOfYears = Integer.parseInt(numberOfYearsString);
22
23      // Enter loan amount
24      String loanString = JOptionPane.showInputDialog(
25        "Enter loan amount, for example 120000.95:");
26
27      // Convert string to double
28      double loanAmount = Double.parseDouble(loanString);
29
30      // Calculate payment
31      double monthlyPayment = loanAmount * monthlyInterestRate / (1
32        - 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
33      double totalPayment = monthlyPayment * numberOfYears * 12;
34
35      // Format to keep two digits after the decimal point
36      monthlyPayment = (int)(monthlyPayment * 100) / 100.0;
37      totalPayment = (int)(totalPayment * 100) / 100.0;
38
39      // Display results
40      String output = "The monthly payment is " + monthlyPayment +
41        "\nThe total payment is " + totalPayment;
42      JOptionPane.showMessageDialog(null, output);
43    }
44  }
```

monthlyPayment

totalPayment

preparing output

The `showInputDialog` method in lines 6–7 displays an input dialog. Enter the interest rate as a double value and click *OK* to accept the input. The value is returned as a string that is assigned to the `String` variable `annualInterestRateString`. The `Double.parseDouble(annualInterestRateString)` (line 11) is used to convert the string into a `double` value. If you entered an input other than a numeric value or clicked *Cancel* in the input dialog box, a runtime error would occur. In Chapter 13, "Exception Handling," you will learn how to handle the exception so that the program can continue to run.



Pedagogical Note

For obtaining input you can use `JOptionPane` or `Scanner`, whichever is convenient. For consistency most examples in this book use `Scanner` for getting input. You can easily revise the examples using `JOptionPane` for getting input.

JOptionPane or **Scanner**?

Key Terms

algorithm 24
assignment operator (=) 30
assignment statement 30
backslash (\) 46
`byte` type 27
casting 41
`char` type 44
constant 31
data type 25
debugger 55
debugging 55
declaration 30
decrement operator (--) 41
`double` type 33
encoding 45
`final` 31

`float` type 35
floating-point number 33
expression 31
identifier 29
increment operator (`++`) 41
incremental development and testing 26
indentation 52
`int` type 34
literal 35
logic error 54
`long` type 35
narrowing (of types) 41
operator 33
overflow 33
pseudocode 30
primitive data type 25
runtime error 54
`short` type 27
syntax error 53
supplementary Unicode 45
underflow 33
Unicode 45
Unix epoch 43
variable 24
widening (of types) 41
whitespace 51

Chapter Summary

1. Identifiers are names for things in a program.
2. An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs ($).
3. An identifier must start with a letter or an underscore. It cannot start with a digit.
4. An identifier cannot be a reserved word.
5. An identifier can be of any length.
6. Choosing descriptive identifiers can make programs easy to read.
7. Variables are used to store data in a program
8. To declare a variable is to tell the compiler what type of data a variable can hold.
9. By convention, variable names are in lowercase.
10. In Java, the equal sign (=) is used as the *assignment operator*.
11. A variable declared in a method must be assigned a value before it can be used.
12. A *named constant* (or simply a *constant*) represents permanent data that never changes.
13. A named constant is declared by using the keyword `final`.
14. By convention, constants are named in uppercase.
15. Java provides four integer types (`byte`, `short`, `int`, `long`) that represent integers of four different sizes.

16. Java provides two floating-point types (`float`, `double`) that represent floating-point numbers of two different precisions.

17. Java provides operators that perform numeric operations: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (remainder).

18. Integer arithmetic (`/`) yields an integer result.

19. The numeric operators in a Java expression are applied the same way as in an arithmetic expression.

20. Java provides shorthand operators `+=` (addition assignment), `-=` (subtraction assignment), `*=` (multiplication assignment), `/=` (division assignment), and `%=` (remainder assignment).

21. The increment operator (`++`) and the decrement operator (`--`) increment or decrement a variable by `1`.

22. When evaluating an expression with values of mixed types, Java automatically converts the operands to appropriate types.

23. You can explicitly convert a value from one type to another using the `(type)exp` notation.

24. Casting a variable of a type with a small range to a variable of a type with a larger range is known as *widening a type*.

25. Casting a variable of a type with a large range to a variable of a type with a smaller range is known as *narrowing a type*.

26. Widening a type can be performed automatically without explicit casting. Narrowing a type must be performed explicitly.

27. Character type (`char`) represents a single character.

28. The character `\` is called the escape character.

29. Java allows you to use escape sequences to represent special characters such as `'\t'` and `'\n'`.

30. The characters `' '`, `'\t'`, `'\f'`, `'\r'`, and `'\n'` are known as the whitespace characters.

31. In computer science, midnight of January 1, 1970, is known as the *Unix epoch.*

32. Programming errors can be categorized into three types: syntax errors, runtime errors, and logic errors.

33. Errors that occur during compilation are called *syntax errors* or *compile errors.*

34. *Runtime errors* are errors that cause a program to terminate abnormally.

35. *Logic errors* occur when a program does not perform the way it was intended to.

Review Questions

Sections 2.2–2.7

2.1 Which of the following identifiers are valid? Which are Java keywords?

```
applet, Applet, a++, --a, 4#R, $4, #44, apps
class, public, int, x, y, radius
```

2.2 Translate the following algorithm into Java code:

- Step 1: Declare a `double` variable named `miles` with initial value `100`;
- Step 2: Declare a `double` constant named `MILES_PER_KILOMETER` with value `1.609`;
- Step 3: Declare a `double` variable named `kilometers`, multiply miles and `MILES_PER_KILOMETER`, and assign the result to `kilometers`.
- Step 4: Display `kilometers` to the console.

What is `kilometers` after Step 4?

2.3 What are the benefits of using constants? Declare an `int` constant `SIZE` with value `20`.

Sections 2.8–2.10

2.4 Assume that `int a = 1` and `double d = 1.0`, and that each expression is independent. What are the results of the following expressions?

```
a = 46 / 9;
a = 46 % 9 + 4 * 4 - 2;
a = 45 + 43 % 5 * (23 * 3 % 2);
a %= 3 / a + 3;
d = 4 + d * d + 4;
d += 1.5 * 3 + (++a);
d -= 1.5 * 3 + a++;
```

2.5 Show the result of the following remainders.

```
56 % 6
78 % -4
-34 % 5
-34 % -5
5 % 1
1 % 5
```

2.6 If today is Tuesday, what will be the day in 100 days?

2.7 Find the largest and smallest `byte`, `short`, `int`, `long`, `float`, and `double`. Which of these data types requires the least amount of memory?

2.8 What is the result of `25 / 4`? How would you rewrite the expression if you wished the result to be a floating-point number?

2.9 Are the following statements correct? If so, show the output.

```
System.out.println("25 / 4 is " + 25 / 4);
System.out.println("25 / 4.0 is " + 25 / 4.0);
System.out.println("3 * 2 / 4 is " + 3 * 2 / 4);
System.out.println("3.0 * 2 / 4 is " + 3.0 * 2 / 4);
```

2.10 How would you write the following arithmetic expression in Java?

$$\frac{4}{3(r+34)} - 9(a+bc) + \frac{3+d(2+a)}{a+bd}$$

2.11 Suppose `m` and `r` are integers. Write a Java expression for mr^2 to obtain a floating-point result.

2.12 Which of these statements are true?

(a) Any expression can be used as a statement.
(b) The expression `x++` can be used as a statement.
(c) The statement `x = x + 5` is also an expression.
(d) The statement `x = y = x = 0` is illegal.

2.13 Which of the following are correct literals for floating-point numbers?

```
12.3, 12.3e+2, 23.4e-2, -334.4, 20, 39F, 40D
```

2.14 Identify and fix the errors in the following code:

```
1 public class Test {
2   public void main(string[] args) {
3     int i;
4     int k = 100.0;
5     int j = i + 1;
6
7     System.out.println("j is " + j + " and
8       k is " + k);
9   }
10 }
```

2.15 How do you obtain the current minute using the `System.currentTimeMillis()` method?

Section 2.11

2.16 Can different types of numeric values be used together in a computation?

2.17 What does an explicit conversion from a `double` to an `int` do with the fractional part of the *double* value? Does casting change the variable being cast?

2.18 Show the following output.

```
float f = 12.5F;
int i = (int)f;
System.out.println("f is " + f);
System.out.println("i is " + i);
```

Section 2.13

2.19 Use print statements to find out the ASCII code for `'1'`, `'A'`, `'B'`, `'a'`, `'b'`. Use print statements to find out the character for the decimal code `40`, `59`, `79`, `85`, `90`. Use print statements to find out the character for the hexadecimal code `40`, `5A`, `71`, `72`, `7A`.

2.20 Which of the following are correct literals for characters?

```
'1', '\u345dE', '\u3fFa', '\b', \t
```

2.21 How do you display characters `\` and `"`?

2.22 Evaluate the following:

```
int i = '1';
int j = '1' + '2';
int k = 'a';
char c = 90;
```

2.23 Can the following conversions involving casting be allowed? If so, find the converted result.

```
char c = 'A';
i = (int)c;
float f = 1000.34f;
int i = (int)f;

double d = 1000.34;
int i = (int)d;

int i = 97;
char c = (char)i;
```

2.24 Show the output of the following program:

```
public class Test {
  public static void main(String[] args) {
    char x = 'a';
    char y = 'c';

    System.out.println(++x);
    System.out.println(y++);
    System.out.println(x - y);
  }
}
```

Section 2.15

2.25 Show the output of the following statements (write a program to verify your result):

```
System.out.println("1" + 1);
System.out.println('1' + 1);
System.out.println("1" + 1 + 1);
System.out.println("1" + (1 + 1));
System.out.println('1' + 1 + 1);
```

2.26 Evaluate the following expressions (write a program to verify your result):

```
1 + "Welcome " + 1 + 1
1 + "Welcome " + (1 + 1)
1 + "Welcome " + ('\u0001'  + 1)
1 + "Welcome " + 'a'  + 1
```

Sections 2.16–2.17

2.27 What are the naming conventions for class names, method names, constants, and variables? Which of the following items can be a constant, a method, a variable, or a class according to the Java naming conventions?

```
MAX_VALUE, Test, read, readInt
```

2.28 Reformat the following program according to the programming style and documentation guidelines. Use the next-line brace style.

```
public class Test
{
  // Main method
  public static void main(String[] args) {
```

```
    /** Print a line */
    System.out.println("2 % 3 = "+2%3);
    }
}
```

2.29 Describe syntax errors, runtime errors, and logic errors.

Section 2.18

2.30 Why do you have to import `JOptionPane` but not the `Math` class?

2.31 How do you prompt the user to enter an input using a dialog box?

2.32 How do you convert a string to an integer? How do you convert a string to a double?

Programming Exercises



Note

Students can run all exercises by downloading `exercise8e.zip` from `www.cs.armstrong.edu/liang/intro8e/exercise8e.zip` and use the command `java -cp exercise8e.zip Exercisei_j` to run Exercise*i_j*. For example, to run Exercise2_1, use

sample runs

```
java -cp exercise8e.zip Exercise2_1
```

This will give you an idea how the program runs.



Debugging TIP

The compiler usually gives a reason for a syntax error. If you don't know how to correct it, compare your program closely, character by character, with similar examples in the text.

learn from examples

Sections 2.2–2.9

2.1 (*Converting Celsius to Fahrenheit*) Write a program that reads a Celsius degree in double from the console, then converts it to Fahrenheit and displays the result. The formula for the conversion is as follows:

```
fahrenheit = (9 / 5) * celsius + 32
```

Hint: In Java, `9 / 5` is `1`, but `9.0 / 5` is `1.8`.

Here is a sample run:

```
Enter a degree in Celsius: 43 Enter
43 Celsius is 109.4 Fahrenheit
```



2.2 (*Computing the volume of a cylinder*) Write a program that reads in the radius and length of a cylinder and computes volume using the following formulas:

```
area = radius * radius * π
volume = area * length
```

Here is a sample run:



```
Enter the radius and length of a cylinder: 5.5 12 Enter
The area is 95.0331
The volume is 1140.4
```

2.3 (*Converting feet into meters*) Write a program that reads a number in feet, converts it to meters, and displays the result. One foot is **0.305** meter. Here is a sample run:



```
Enter a value for feet: 16 Enter
16 feet is 4.88 meters
```

2.4 (*Converting pounds into kilograms*) Write a program that converts pounds into kilograms. The program prompts the user to enter a number in pounds, converts it to kilograms, and displays the result. One pound is **0.454** kilograms. Here is a sample run:



```
Enter a number in pounds: 55.5 Enter
55.5 pounds is 25.197 kilograms
```

2.5* (*Financial application: calculating tips*) Write a program that reads the subtotal and the gratuity rate, then computes the gratuity and total. For example, if the user enters **10** for subtotal and **15%** for gratuity rate, the program displays **$1.5** as gratuity and **$11.5** as total. Here is a sample run:



```
Enter the subtotal and a gratuity rate: 15.69 15 Enter
The gratuity is 2.35 and total is 18.04
```

2.6** (*Summing the digits in an integer*) Write a program that reads an integer between **0** and **1000** and adds all the digits in the integer. For example, if an integer is **932**, the sum of all its digits is **14**.

Hint: Use the **%** operator to extract digits, and use the **/** operator to remove the extracted digit. For instance, **932 % 10 = 2** and **932 / 10 = 93**.

Here is a sample run:



```
Enter a number between 0 and 1000: 999 Enter
The sum of the digits is 27
```

2.7* (*Finding the number of years*) Write a program that prompts the user to enter the minutes (e.g., 1 billion) and displays the number of years and days for the minutes. For simplicity, assume a year has **365** days. Here is a sample run:



```
Enter the number of minutes: 1000000000 Enter
1000000000 minutes is approximately 1902 years and 214 days.
```

Section 2.13

2.8* (*Finding the character of an ASCII code*) Write a program that receives an ASCII code (an integer between 0 and 128) and displays its character. For example, if the user enters 97, the program displays character a. Here is a sample run:

```
Enter an ASCII code: 69 Enter
The character for ASCII code 69 is E
```



2.9* (*Financial application: monetary units*) Rewrite Listing 2.10, ComputeChange.java, to fix the possible loss of accuracy when converting a `double` value to an `int` value. Enter the input as an integer whose last two digits represent the cents. For example, the input 1156 represents 11 dollars and 56 cents.

Section 2.18

2.10* (*Using the GUI input*) Rewrite Listing 2.10, ComputeChange.java, using the GUI input and output.

Comprehensive

2.11* (*Financial application: payroll*) Write a program that reads the following information and prints a payroll statement:

Employee's name (e.g., Smith)

Number of hours worked in a week (e.g., 10)

Hourly pay rate (e.g., 6.75)

Federal tax withholding rate (e.g., 20%)

State tax withholding rate (e.g., 9%)

Write this program in two versions: (a) Use dialog boxes to obtain input and display output; (b) Use console input and output. A sample run of the console input and output is shown below:

```
Enter employee's name: Smith Enter
Enter number of hours worked in a week: 10 Enter
Enter hourly pay rate: 6.75 Enter
Enter federal tax withholding rate: 0.20 Enter
Enter state tax withholding rate: 0.09 Enter
Employee Name: Smith
Hours Worked:  10.0
Pay Rate:  $6.75
Gross Pay:  $67.5
Deductions:
  Federal Withholding (20.0%):  $13.5
  State Withholding (9.0%):  $6.07
  Total Deduction:  $19.57
Net Pay:  $47.92
```



2.12* (*Financial application: calculating interest*) If you know the balance and the annual percentage interest rate, you can compute the interest on the next monthly payment using the following formula:

$$\text{interest} = \text{balance} \times (\text{annualInterestRate} / 1200)$$

Write a program that reads the balance and the annual percentage interest rate and displays the interest for the next month in two versions: (a) Use dialog boxes to obtain input and display output; (b) Use console input and output. Here is a sample run:



```
Enter balance and interest rate (e.g., 3 for 3%): 1000 3.5 Enter
The interest is 2.91667
```

2.13* (*Financial application: calculating the future investment value*) Write a program that reads in investment amount, annual interest rate, and number of years, and displays the future investment value using the following formula:

```
futureInvestmentValue =
  investmentAmount x (1 + monthlyInterestRate)^numberOfYears*12
```

For example, if you enter amount **1000**, annual interest rate **3.25%**, and number of years **1**, the future investment value is **1032.98**.

Hint: Use the **Math.pow(a, b)** method to compute **a** raised to the power of **b**.

Here is a sample run:



```
Enter investment amount: 1000 Enter
Enter monthly interest rate: 4.25 Enter
Enter number of years: 1 Enter
Accumulated value is 1043.34
```

2.14* (*Health application: computing BMI*) Body Mass Index (BMI) is a measure of health on weight. It can be calculated by taking your weight in kilograms and dividing by the square of your height in meters. Write a program that prompts the user to enter a weight in pounds and height in inches and display the BMI. Note that one pound is **0.45359237** kilograms and one inch is **0.0254** meters. Here is a sample run:



Video Note
Compute BMI



```
Enter weight in pounds: 95.5 Enter
Enter height in inches: 50 Enter
BMI is 26.8573
```

2.15** (*Financial application: compound value*) Suppose you save **$100** *each* month into a savings account with the annual interest rate 5%. So, the monthly interest rate is 0.05 / 12 = 0.00417. After the first month, the value in the account becomes

```
100 * (1 + 0.00417) = 100.417
```

After the second month, the value in the account becomes

```
(100 + 100.417) * (1 + 0.00417) = 201.252
```

After the third month, the value in the account becomes

```
(100 + 201.252) * (1 + 0.00417) = 302.507
```

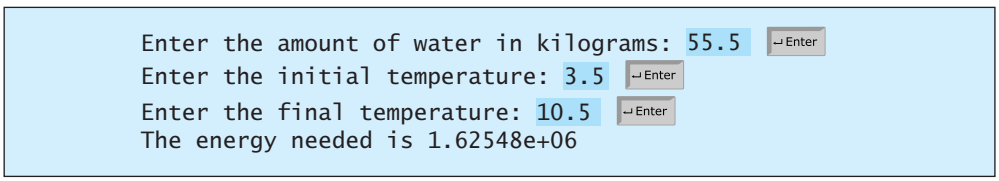
and so on.

Write a program to display the account value after the sixth month. (In Exercise 4.30, you will use a loop to simplify the code and display the account value for any month.)

2.16 (*Science: calculating energy*) Write a program that calculates the energy needed to heat water from an initial temperature to a final temperature. Your program should prompt the user to enter the amount of water in kilograms and the initial and final temperatures of the water. The formula to compute the energy is

```
Q = M * (final temperature - initial temperature) * 4184
```

where `M` is the weight of water in kilograms, temperatures are in degrees Celsius, and energy `Q` is measured in joules. Here is a sample run:

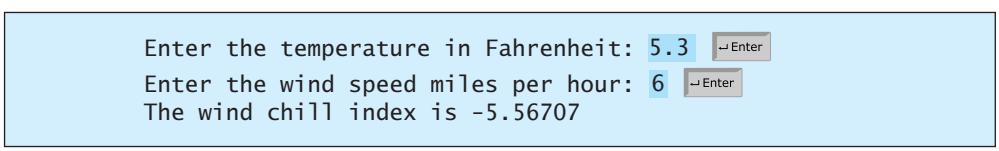


2.17* (*Science: wind-chill temperature*) How cold is it outside? The temperature alone is not enough to provide the answer. Other factors including wind speed, relative humidity, and sunshine play important roles in determining coldness outside. In 2001, the National Weather Service (NWS) implemented the new wind-chill temperature to measure the coldness using temperature and wind speed. The formula is given as follows:

$$t_{wc} = 35.74 + 0.6215t_a - 35.75v^{0.16} + 0.4275t_a v^{0.16}$$

where t_a is the outside temperature measured in degrees Fahrenheit and v is the speed measured in miles per hour. t_{wc} is the wind-chill temperature. The formula cannot be used for wind speeds below 2 mph or temperatures below −58°F or above 41°F.

Write a program that prompts the user to enter a temperature between −58°F and 41°F and a wind speed greater than or equal to `2` and displays the wind-chill temperature. Use `Math.pow(a, b)` to compute $v^{0.16}$. Here is a sample run:



2.18 (*Printing a table*) Write a program that displays the following table:

```
a     b     pow(a, b)
1     2     1
2     3     8
3     4     81
4     5     1024
5     6     15625
```

2.19 (*Random character*) Write a program that displays a random uppercase letter using the `System.CurrentTimeMillis()` method.

2.20 (*Geometry: distance of two points*) Write a program that prompts the user to enter two points `(x1, y1)` and `(x2, y2)` and displays their distances. The formula for computing the distance is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Note you can use the `Math.pow(a, 0.5)` to compute $\sqrt{a}$. Here is a sample run:



```
Enter x1 and y1: 1.5 -3.4 Enter
Enter x2 and y2: 4 5 Enter
The distance of the two points is 8.764131445842194
```

2.21* (*Geometry: area of a triangle*) Write a program that prompts the user to enter three points `(x1, y1)`, `(x2, y2)`, `(x3, y3)` of a triangle and displays its area. The formula for computing the area of a triangle is

$$s = (side1 + side2 + side3)/2;$$

$$area = \sqrt{s(s - side1)(s - side2)(s - side3)}$$

Here is a sample run.



```
Enter three points for a triangle: 1.5 -3.4 4.6 5 9.5 -3.4 Enter
The area of the triangle is 33.6
```

2.22 (*Geometry: area of a hexagon*) Write a program that prompts the user to enter the side of a hexagon and displays its area. The formula for computing the area of a hexagon is

$$Area = \frac{3\sqrt{3}}{2}s^2,$$

where s is the length of a side. Here is a sample run:



```
Enter the side: 5.5 Enter
The area of the hexagon is 78.5895
```

2.23 (*Physics: acceleration*) Average acceleration is defined as the change of velocity divided by the time taken to make the change, as shown in the following formula:

$$a = \frac{v_1 - v_0}{t}$$

Write a program that prompts the user to enter the starting velocity v_0 in meters/second, the ending velocity v_1 in meters/second, and the time span t in seconds, and displays the average acceleration. Here is a sample run:

```
Enter v0, v1, and t: 5.5 50.9 4.5 Enter
The average acceleration is 10.0889
```



2.24 (*Physics: finding runway length*) Given an airplane's acceleration a and take-off speed v, you can compute the minimum runway length needed for an airplane to take off using the following formula:

$$length = \frac{v^2}{2a}$$

Write a program that prompts the user to enter v in meters/second (m/s) and the acceleration a in meters/second squared (m/s^2), and displays the minimum runway length. Here is a sample run:

```
Enter v and a: 60 3.5 Enter
The minimum runway length for this airplane is 514.286
```



2.25* (*Current time*) Listing 2.6, ShowCurrentTime.java, gives a program that displays the current time in GMT. Revise the program so that it prompts the user to enter the time zone offset to GMT and displays the time in the specified time zone. Here is a sample run:

```
Enter the time zone offset to GMT: -5 Enter
The current time is 4:50:34
```



This page intentionally left blank

CHAPTER 3

SELECTIONS

Objectives

- To declare `boolean` type and write Boolean expressions using comparison operators (§3.2).
- To program `AdditionQuiz` using Boolean expressions (§3.3).
- To implement selection control using one-way `if` statements (§3.4)
- To program the `GuessBirthday` game using one-way `if` statements (§3.5).
- To implement selection control using two-way `if` statements (§3.6).
- To implement selection control using nested `if` statements (§3.7).
- To avoid common errors in `if` statements (§3.8).
- To program using selection statements for a variety of examples(`SubtractionQuiz`, `BMI`, `ComputeTax`) (§3.9–3.11).
- To generate random numbers using the `Math.random()` method (§3.9).
- To combine conditions using logical operators (`&&`, `||`, and `!`) (§3.12).
- To program using selection statements with combined conditions (`LeapYear`, `Lottery`) (§§3.13–3.14).
- To implement selection control using `switch` statements (§3.15).
- To write expressions using the conditional operator (§3.16).
- To format output using the `System.out.printf` method and to format strings using the `String.format` method (§3.17).
- To examine the rules governing operator precedence and associativity (§3.18).
- (GUI) To get user confirmation using confirmation dialogs (§3.19).



3.1 Introduction

problem

If you enter a negative value for `radius` in Listing 2.2, ComputeAreaWithConsoleInput.java, the program prints an invalid result. If the radius is negative, you don't want the program to compute the area. How can you deal with this situation?

Like all high-level programming languages, Java provides selection statements that let you choose actions with two or more alternative courses. You can use the following selection statement to replace lines 12–17 in Listing 2.2:

```
if (radius < 0)
  System.out.println("Incorrect input");
else {
  area = radius * radius * 3.14159;
  System.out.println("Area is " + area);
}
```

Selection statements use conditions. Conditions are Boolean expressions. This chapter first introduces Boolean types, values, comparison operators, and expressions.

3.2 `boolean` Data Type

comparison operators

How do you compare two values, such as whether a radius is greater than `0`, equal to `0`, or less than `0`? Java provides six *comparison operators* (also known as *relational operators*), shown in Table 3.1, which can be used to compare two values (assume radius is `5` in the table).

TABLE 3.1 Comparison Operators

Operator	*Name*	*Example*	*Result*
`<`	less than	`radius < 0`	`false`
`<=`	less than or equal to	`radius <= 0`	`false`
`>`	greater than	`radius > 0`	`true`
`>=`	greater than or equal to	`radius >= 0`	`true`
`==`	equal to	`radius == 0`	`false`
`!=`	not equal to	`radius != 0`	`true`



Note

compare characters

You can also compare characters. Comparing characters is the same as comparing their Unicodes. For example, `'a'` is larger than `'A'` because the Unicode of `'a'` is larger than the Unicode of `'A'`. See Appendix B, "The ASCII Character Sets," to find the order of characters.



Caution

== vs. =

The equality comparison operator is two equal signs (`==`), not a single equal sign (`=`). The latter symbol is for assignment.

The result of the comparison is a Boolean value: `true` or `false`. For example, the following statement displays `true`:

```
double radius = 1;
System.out.println(radius > 0);
```

Boolean variable

A variable that holds a Boolean value is known as a *Boolean variable*. The `boolean` data type is used to declare Boolean variables. A `boolean` variable can hold one of the two values:

`true` and `false`. For example, the following statement assigns `true` to the variable `lightsOn`:

```
boolean lightsOn = true;
```

`true` and `false` are literals, just like a number such as `10`. They are reserved words and cannot be used as identifiers in your program.

Boolean literals

3.3 Problem: A Simple Math Learning Tool



Video Note
Program addition quiz

Suppose you want to develop a program to let a first-grader practice addition. The program randomly generates two single-digit integers, `number1` and `number2`, and displays to the student a question such as "What is 7 + 9?", as shown in the sample run. After the student types the answer, the program displays a message to indicate whether it is true or false.

There are several ways to generate random numbers. For now, generate the first integer using `System.currentTimeMillis() % 10` and the second using `System.currentTimeMillis() * 7 % 10`. Listing 3.1 gives the program. Lines 5–6 generate two numbers, `number1` and `number2`. Line 14 obtains an answer from the user. The answer is graded in line 18 using a Boolean expression `number1 + number2 == answer`.

LISTING 3.1 AdditionQuiz.java

```
1 import java.util.Scanner;
2
3 public class AdditionQuiz {
4   public static void main(String[] args) {
5     int number1 = (int)(System.currentTimeMillis() % 10);
6     int number2 = (int)(System.currentTimeMillis() * 7 % 10);
7
8     // Create a Scanner
9     Scanner input = new Scanner(System.in);
10
11    System.out.print(
12      "What is " + number1 + " + " + number2 + "? ");
13
14    int answer = input.nextInt();
15
16    System.out.println(
17      number1 + " + " + number2 + " = " + answer + " is " +
18      (number1 + number2 == answer));
19  }
20 }
```

generate number1
generate number2

show question

display result

```
What is 1 + 7? 8 Enter
1 + 7 = 8 is true
```



```
What is 4 + 8? 9 Enter
4 + 8 = 9 is false
```





line#	number1	number2	answer	output
5	4			
6		8		
14			9	
16				4 + 8 = 9 is false

3.4 `if` Statements

why `if` statement?

The preceding program displays a message such as "6 + 2 = 7 is false." If you wish the message to be "6 + 2 = 7 is incorrect," you have to use a selection statement to carry out this minor change.

This section introduces selection statements. Java has several types of selection statements: one-way `if` statements, two-way `if` statements, nested `if` statements, `switch` statements, and conditional expressions.

3.4.1 One-Way `if` Statements

A one-way `if` statement executes an action if and only if the condition is `true`. The syntax for a one-way `if` statement is shown below:

`if` statement

```
if (boolean-expression) {
  statement(s);
}
```

The execution flow chart is shown in Figure 3.1(a).

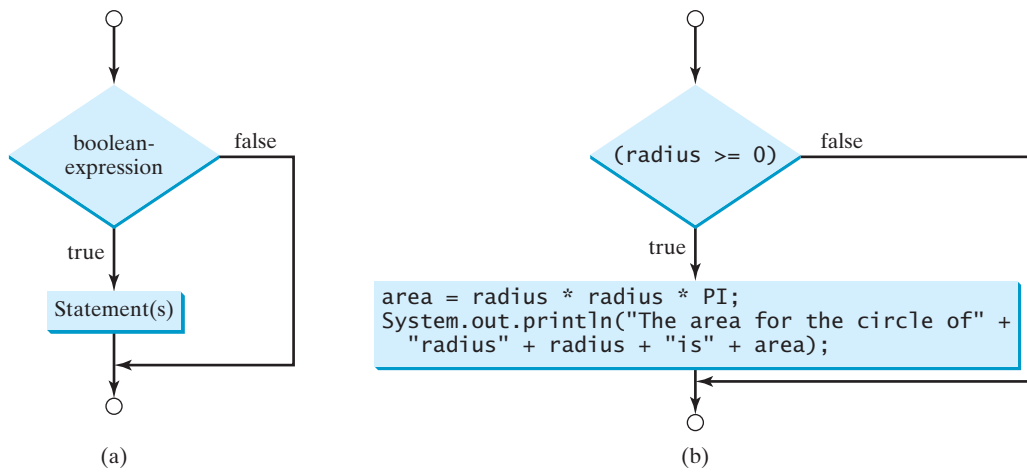


FIGURE 3.1 An `if` statement executes statements if the `boolean-expression` evaluates to `true`.

If the `boolean-expression` evaluates to `true`, the statements in the block are executed. As an example, see the following code:

```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("The area for the circle of radius " +
    radius + " is " + area);
}
```

The flow chart of the preceding statement is shown in Figure 3.1(b). If the value of `radius` is greater than or equal to `0`, then the `area` is computed and the result is displayed; otherwise, the two statements in the block will not be executed.

The `boolean-expression` is enclosed in parentheses. For example, the code in (a) below is wrong. It should be corrected, as shown in (b).

```
if i > 0 {
  System.out.println("i is positive");
}
```

(a) Wrong

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(b) Correct

The block braces can be omitted if they enclose a single statement. For example, the following statements are equivalent.

```
if (i > 0) {
  System.out.println("i is positive");
}
```

(a)

Equivalent

```
if (i > 0)
  System.out.println("i is positive");
```

(b)

Listing 3.2 gives a program that prompts the user to enter an integer. If the number is a multiple of 5, print HiFive. If the number is divisible by 2, print HiEven.

LISTING 3.2 SimpleIfDemo.java

```
1 import java.util.Scanner;
2
3 public class SimpleIfDemo {
4   public static void main(String[] args) {
5     Scanner input = new Scanner(System.in);
6     System.out.println("Enter an integer: ");
7     int number = input.nextInt();                    enter input
8
9     if (number % 5 == 0)                             check 5
10      System.out.println("HiFive");
11
12    if (number % 2 == 0)                             check even
13      System.out.println("HiEven");
14  }
15 }
```

```
Enter an integer: 4 Enter
HiEven
```



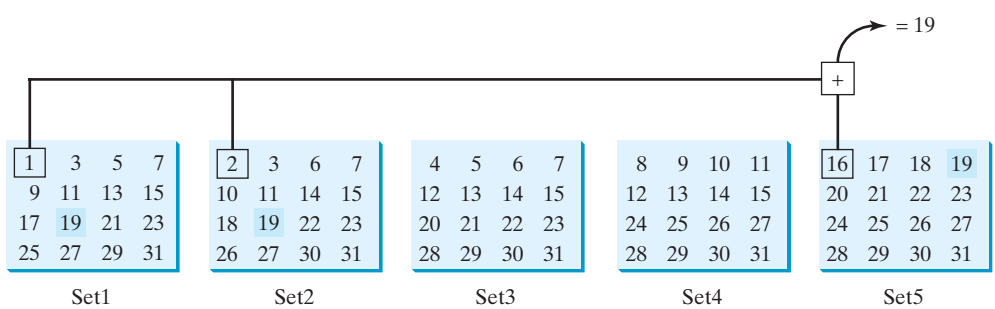
```
Enter an integer: 30 Enter
HiFive
HiEven
```



The program prompts the user to enter an integer (line 7) and displays HiFive if it is divisible by 5 (lines 9–10) and HiEven if it is divisible by 2 (lines 12–13).

3.5 Problem: Guessing Birthdays

You can find out the date of the month when your friend was born by asking five questions. Each question asks whether the day is in one of the five sets of numbers.



The birthday is the sum of the first numbers in the sets where the day appears. For example, if the birthday is **19**, it appears in Set1, Set2, and Set5. The first numbers in these three sets are **1**, **2**, and **16**. Their sum is **19**.

Listing 3.3 gives a program that prompts the user to answer whether the day is in Set1 (lines 41–47), in Set2 (lines 50–56), in Set3 (lines 59–65), in Set4 (lines 68–74), and in Set5 (lines 77–83). If the number is in the set, the program adds the first number in the set to **day** (lines 47, 56, 65, 74, 83).

LISTING 3.3 GuessBirthday.java

```
1  import java.util.Scanner;
2
3  public class GuessBirthday {
4    public static void main(String[] args) {
5      String set1 =
6        " 1  3  5  7\n" +
7        " 9 11 13 15\n" +
8        "17 19 21 23\n" +
9        "25 27 29 31";
10
11     String set2 =
12       " 2  3  6  7\n" +
13       "10 11 14 15\n" +
14       "18 19 22 23\n" +
15       "26 27 30 31";
16
17     String set3 =
18       " 4  5  6  7\n" +
19       "12 13 14 15\n" +
20       "20 21 22 23\n" +
21       "28 29 30 31";
22
23     String set4 =
24       " 8  9 10 11\n" +
25       "12 13 14 15\n" +
26       "24 25 26 27\n" +
27       "28 29 30 31";
28
29     String set5 =
30       "16 17 18 19\n" +
31       "20 21 22 23\n" +
32       "24 25 26 27\n" +
33       "28 29 30 31";
34
35     int day = 0;
36
37     // Create a Scanner
38     Scanner input = new Scanner(System.in);
39
40     // Prompt the user to answer questions
41     System.out.print("Is your birthday in Set1?\n");
42     System.out.print(set1);
43     System.out.print("\nEnter 0 for No and 1 for Yes: ");
44     int answer = input.nextInt();
45
46     if (answer == 1)
47       day += 1;
48
```

day to be determined

in Set1?

```
49      // Prompt the user to answer questions
50      System.out.print("\nIs your birthday in Set2?\n" );
51      System.out.print(set2);
52      System.out.print("\nEnter 0 for No and 1 for Yes: ");
53      answer = input.nextInt();
54
55      if (answer == 1)                                          in Set2?
56        day += 2;
57
58      // Prompt the user to answer questions
59      System.out.print("Is your birthday in Set3?\n");
60      System.out.print(set3);
61      System.out.print("\nEnter 0 for No and 1 for Yes: ");
62      answer = input.nextInt();
63
64      if (answer == 1)                                          in Set3?
65        day += 4;
66
67      // Prompt the user to answer questions
68      System.out.print("\nIs your birthday in Set4?\n");
69      System.out.print(set4);
70      System.out.print("\nEnter 0 for No and 1 for Yes: ");
71      answer = input.nextInt();
72
73      if (answer == 1)                                          in Set4?
74        day += 8;
75
76      // Prompt the user to answer questions
77      System.out.print("\nIs your birthday in Set5?\n");
78      System.out.print(set5);
79      System.out.print("\nEnter 0 for No and 1 for Yes: ");
80      answer = input.nextInt();
81
82      if (answer == 1)                                          in Set5?
83        day += 16;
84
85      System.out.println("\nYour birthday is " + day + "!");
86    }
87 }
```

```
Is your birthday in Set1?
 1  3  5  7
 9 11 13 15
17 19 21 23
25 27 29 31
Enter 0 for No and 1 for Yes: 1 ↵Enter
Is your birthday in Set2?
 2  3  6  7
10 11 14 15
18 19 22 23
26 27 30 31
Enter 0 for No and 1 for Yes: 1 ↵Enter
Is your birthday in Set3?
 4  5  6  7
12 13 14 15
20 21 22 23
28 29 30 31
Enter 0 for No and 1 for Yes: 0 ↵Enter
```



```
Is your birthday in Set4?
 8  9 10 11
12 13 14 15
24 25 26 27
28 29 30 31
Enter 0 for No and 1 for Yes: 0 Enter

Is your birthday in Set5?
16 17 18 19
20 21 22 23
24 25 26 27
28 29 30 31
Enter 0 for No and 1 for Yes: 1 Enter
Your birthday is 19
```



line#	day	answer	output
35	0		
44		1	
47	1		
53		1	
56	3		
62		0	
71		0	
80		1	
83	19		
			Your birthday is 19

mathematics behind the game

The game is easy to program. You may wonder how the game was created. The mathematics behind the game is actually quite simple. The numbers are not grouped together by accident. The way they are placed in the five sets is deliberate. The starting numbers in the five sets are **1**, **2**, **4**, **8**, and **16**, which correspond to **1**, **10**, **100**, **1000**, and **10000** in binary. A binary number for decimal integers between **1** and **31** has at most five digits, as shown in Figure 3.2(a). Let it be $b_5b_4b_3b_2b_1$. So, $b_5b_4b_3b_2b_1 = b_5\,0000 + b_4\,000 + b_3\,00 + b_2\,0 + b_1$, as shown in Figure 3.2(b). If a day's binary number has a digit **1** in b_k, the number should appear in Set*k*. For example, number **19** is binary **10011**, so it appears in Set1, Set2, and Set5. It is binary **1 + 10 + 10000 = 10011** or decimal **1 + 2 + 16 = 19**. Number **31** is binary **11111**, so it appears in Set1, Set2, Set3, Set4, and Set5. It is binary **1 + 10 + 100 + 1000 + 10000 = 11111** or decimal **1 + 2 + 4 + 8 + 16 = 31**.

Decimal	Binary
1	00001
2	00010
3	00011
…	
19	10011
…	
31	11111

(a)

$$
\begin{array}{r}
b_5\ 0\ 0\ 0\ 0 \\
b_4\ 0\ 0\ 0 \\
b_3\ 0\ 0 \\
b_2\ 0 \\
+\quad b_1 \\
\hline
b_5\,b_4\,b_3\,b_2\,b_1
\end{array}
\qquad
\begin{array}{r}
10000 \\
10 \\
+\quad 1 \\
\hline
10011
\end{array}
\qquad
\begin{array}{r}
10000 \\
1000 \\
100 \\
10 \\
+\quad 1 \\
\hline
11111
\end{array}
$$

19 31

(b)

FIGURE 3.2 (a) A number between **1** and **31** can be represented using a 5-digit binary number. (b) A 5-digit binary number can be obtained by adding binary numbers **1**, **10**, **100**, **1000**, or **10000**.

3.6 Two-Way `if` Statements

A one-way `if` statement takes an action if the specified condition is `true`. If the condition is `false`, nothing is done. But what if you want to take alternative actions when the condition is `false`? You can use a two-way `if` statement. The actions that a two-way `if` statement specifies differ based on whether the condition is `true` or `false`.

Here is the syntax for a two-way `if` statement:

```
if (boolean-expression) {
  statement(s)-for-the-true-case;
}
else {
  statement(s)-for-the-false-case;
}
```

The flow chart of the statement is shown in Figure 3.3.

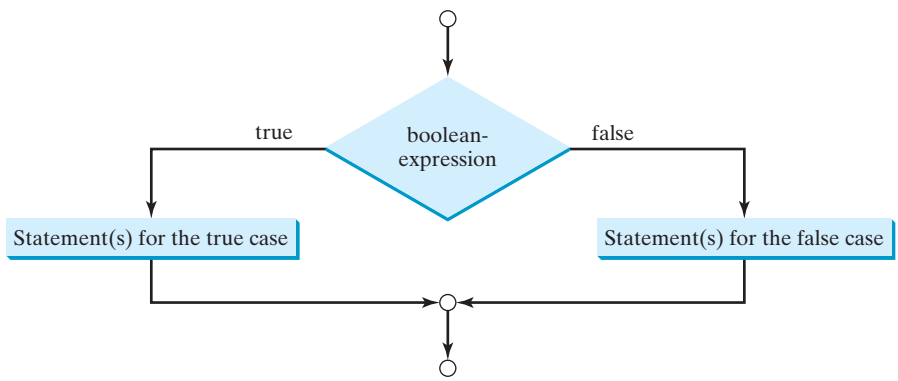


FIGURE 3.3 An `if . . . else` statement executes statements for the true case if the `boolean-expression` evaluates to `true`; otherwise, statements for the false case are executed.

If the `boolean-expression` evaluates to `true`, the statement(s) for the true case are executed; otherwise, the statement(s) for the false case are executed. For example, consider the following code:

two-way `if` statement

```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("The area for the circle of radius " +
    radius + " is " + area);
}
else {
  System.out.println("Negative input");
}
```

If `radius >= 0` is `true`, `area` is computed and displayed; if it is `false`, the message `"Negative input"` is printed.

As usual, the braces can be omitted if there is only one statement within them. The braces enclosing the `System.out.println("Negative input")` statement can therefore be omitted in the preceding example.

Here is another example of using the `if ... else` statement. The example checks whether a number is even or odd, as follows:

```
if (number % 2 == 0)
  System.out.println(number + " is even.");
else
  System.out.println(number + " is odd.");
```

3.7 Nested `if` Statements

The statement in an `if` or `if ... else` statement can be any legal Java statement, including another `if` or `if ... else` statement. The inner `if` statement is said to be *nested* inside the outer `if` statement. The inner `if` statement can contain another `if` statement; in fact, there is no limit to the depth of the nesting. For example, the following is a nested `if` statement:

nested `if` statement

```
if (i > k) {
  if (j > k)
    System.out.println("i and j are greater than k");
}
else
  System.out.println("i is less than or equal to k");
```

The `if (j > k)` statement is nested inside the `if (i > k)` statement.

The nested `if` statement can be used to implement multiple alternatives. The statement given in Figure 3.4(a), for instance, assigns a letter grade to the variable `grade` according to the score, with multiple alternatives.

```
if (score >= 90.0)
  grade = 'A';
else
  if (score >= 80.0)
    grade = 'B';
  else
    if (score >= 70.0)
      grade = 'C';
    else
      if (score >= 60.0)
        grade = 'D';
      else
        grade = 'F';
```

(a)

Equivalent

This is better

```
if (score >= 90.0)
  grade = 'A';
else if (score >= 80.0)
  grade = 'B';
else if (score >= 70.0)
  grade = 'C';
else if (score >= 60.0)
  grade = 'D';
else
  grade = 'F';
```

(b)

FIGURE 3.4 A preferred format for multiple alternative `if` statements is shown in (b).

The execution of this `if` statement proceeds as follows. The first condition `(score >= 90.0)` is tested. If it is `true`, the grade becomes `'A'`. If it is `false`, the second condition `(score >= 80.0)` is tested. If the second condition is `true`, the grade becomes `'B'`. If that condition is `false`, the third condition and the rest of the conditions (if necessary) continue to be tested until a condition is met or all of the conditions prove to be `false`. If all of the conditions are `false`, the grade becomes `'F'`. Note that a condition is tested only when all of the conditions that come before it are `false`.

The `if` statement in Figure 3.4(a) is equivalent to the `if` statement in Figure 3.4(b). In fact, Figure 3.4(b) is the preferred writing style for multiple alternative `if` statements. This style avoids deep indentation and makes the program easy to read.



Tip

Often, to assign a test condition to a **boolean** variable, new programmers write code as in (a) below:

assign **boolean** variable

```
if (number % 2 == 0)
  even = true;
else
  even = false;
```

(a)

Equivalent

This is shorter

```
boolean even
  = number % 2 == 0;
```

(b)

The code can be simplified by assigning the test value directly to the variable, as shown in (b).

3.8 Common Errors in Selection Statements

The following errors are common among new programmers.

Common Error 1: Forgetting Necessary Braces

The braces can be omitted if the block contains a single statement. However, forgetting the braces when they are needed for grouping multiple statements is a common programming error. If you modify the code by adding new statements in an **if** statement without braces, you will have to insert the braces. For example, the code in (a) below is wrong. It should be written with braces to group multiple statements, as shown in (b).

```
if (radius >= 0)
  area = radius * radius * PI;
  System.out.println("The area "
    + " is " + area);
```

(a) Wrong

```
if (radius >= 0) {
  area = radius * radius * PI;
  System.out.println("The area "
    + " is " + area);
}
```

(b) Correct

Common Error 2: Wrong Semicolon at the if Line

Adding a semicolon at the **if** line, as shown in (a) below, is a common mistake.

Logic Error

```
if (radius >= 0);
{
  area = radius * radius * PI;
  System.out.println("The area "
    + " is " + area);
}
```

(a)

Equivalent

Empty Block

```
if (radius >= 0) { };
{
  area = radius * radius * PI;
  System.out.println("The area "
    + " is " + area);
}
```

(b)

This mistake is hard to find, because it is neither a compilation error nor a runtime error; it is a logic error. The code in (a) is equivalent to that in (b) with an empty block.

This error often occurs when you use the next-line block style. Using the end-of-line block style can help prevent the error.

Common Error 3: Redundant Testing of Boolean Values

To test whether a **boolean** variable is **true** or **false** in a test condition, it is redundant to use the equality comparison operator like the code in (a):

```
if (even == true)
  System.out.println(
    "It is even.");
```

(a)

Equivalent

This is better

```
if (even)
  System.out.println(
    "It is even.");
```

(b)

Instead, it is better to test the `boolean` variable directly, as shown in (b). Another good reason for doing this is to avoid errors that are difficult to detect. Using the `=` operator instead of the `==` operator to compare equality of two items in a test condition is a common error. It could lead to the following erroneous statement:

```
if (even = true)
  System.out.println("It is even.");
```

This statement does not have syntax errors. It assigns `true` to `even`, so that `even` is always `true`.

Common Error 4: Dangling `else` Ambiguity

The code in (a) below has two `if` clauses and one `else` clause. Which `if` clause is matched by the `else` clause? The indentation indicates that the `else` clause matches the first `if` clause. However, the `else` clause actually matches the second `if` clause. This situation is known as the *dangling-else ambiguity*. The `else` clause always matches the most recent unmatched `if` clause in the same block. So, the statement in (a) is equivalent to the code in (b).

```
int i = 1;
int j = 2;
int k = 3;

if (i > j)
  if (i > k)
    System.out.println("A");
else
    System.out.println("B");
```

(a)

Equivalent

This is better with correct indentation →

```
int i = 1;
int j = 2;
int k = 3;

if (i > j)
  if (i > k)
    System.out.println("A");
  else
    System.out.println("B");
```

(b)

Since `(i > j)` is false, nothing is printed from the statement in (a) and (b). To force the `else` clause to match the first `if` clause, you must add a pair of braces:

```
int i = 1, j = 2, k = 3;

if (i > j) {
  if (i > k)
    System.out.println("A");
}
else
  System.out.println("B");
```

This statement prints `B`.

3.9 Problem: An Improved Math Learning Tool



Video Note
Program subtraction quiz

Suppose you want to develop a program for a first-grader to practice subtraction. The program randomly generates two single-digit integers, `number1` and `number2`, with `number1 >= number2` and displays to the student a question such as "What is 9 − 2?" After the student enters the answer, the program displays a message indicating whether it is correct.

`random()` method

The previous programs generate random numbers using `System.currentTimeMillis()`. A better approach is to use the `random()` method in the `Math` class. Invoking this method returns a random double value `d` such that $0.0 \le d < 1.0$. So, `(int)(Math.random() * 10)` returns a random single-digit integer (i.e., a number between 0 and 9).

The program may work as follows:

- Generate two single-digit integers into `number1` and `number2`.
- If `number1 < number2`, swap `number1` with `number2`.

- Prompt the student to answer "What is number1 – number2?"
- Check the student's answer and display whether the answer is correct.

The complete program is shown in Listing 3.4.

LISTING 3.4 SubtractionQuiz.java

```
1 import java.util.Scanner;
2
3 public class SubtractionQuiz {
4   public static void main(String[] args) {
5     // 1. Generate two random single-digit integers
6     int number1 = (int)(Math.random() * 10);                    random numbers
7     int number2 = (int)(Math.random() * 10);
8
9     // 2. If number1 < number2, swap number1 with number2
10    if (number1 < number2) {
11      int temp = number1;
12      number1 = number2;
13      number2 = temp;
14    }
15
16    // 3. Prompt the student to answer "What is number1 - number2?"
17    System.out.print
18      ("What is " + number1 + " - " + number2 + "? ");
19    Scanner input = new Scanner(System.in);
20    int answer = input.nextInt();                               get answer
21
22    // 4. Grade the answer and display the result
23    if (number1 - number2 == answer)                            check the answer
24      System.out.println("You are correct!");
25    else
26      System.out.println("Your answer is wrong\n" + number1 + " - "
27        + number2 + " should be " + (number1 - number2));
28  }
29 }
```

```
What is 6 - 6? 0 Enter
You are correct!
```



```
What is 9 - 2? 5 Enter
Your answer is wrong
9 - 2 should be 7
```





line#	number1	number2	temp	answer	output
6	2				
7		9			
11			2		
12	9				
13		2			
20				5	
26					Your answer is wrong 9 – 2 should be 7

To swap two variables `number1` and `number2`, a temporary variable `temp` (line 11) is used to first hold the value in `number1`. The value in `number2` is assigned to `number1` (line 12), and the value in `temp` is assigned to `number2` (line 13).

3.10 Problem: Computing Body Mass Index

Body Mass Index (BMI) is a measure of health on weight. It can be calculated by taking your weight in kilograms and dividing by the square of your height in meters. The interpretation of BMI for people 16 years or older is as follows:

BMI	Interpretation
below 16	seriously underweight
16–18	underweight
18–24	normal weight
24–29	overweight
29–35	seriously overweight
above 35	gravely overweight

Write a program that prompts the user to enter a weight in pounds and height in inches and display the BMI. Note that one pound is `0.45359237` kilograms and one inch is `0.0254` meters. Listing 3.5 gives the program.

LISTING 3.5 ComputeBMI.java

```
1  import java.util.Scanner;
2
3  public class ComputeAndInterpretBMI {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6
7      // Prompt the user to enter weight in pounds
8      System.out.print("Enter weight in pounds: ");
9      double weight = input.nextDouble();                    input weight
10
11     // Prompt the user to enter height in inches
12     System.out.print("Enter height in inches: ");
13     double height = input.nextDouble();                    input height
14
15     final double KILOGRAMS_PER_POUND = 0.45359237; // Constant
16     final double METERS_PER_INCH = 0.0254; // Constant
17
18     // Compute BMI
19     double weightInKilograms = weight * KILOGRAMS_PER_POUND;
20     double heightInMeters = height * METERS_PER_INCH;
21     double bmi = weightInKilograms /                       compute bmi
22       (heightInMeters * heightInMeters);
23
24     // Display result
25     System.out.printf("Your BMI is %5.2f\n", bmi);         display output
26     if (bmi < 16)
27       System.out.println("You are seriously underweight");
28     else if (bmi < 18)
29       System.out.println("You are underweight");
30     else if (bmi < 24)
31       System.out.println("You are normal weight");
```

```
32      else if (bmi < 29)
33        System.out.println("You are overweight");
34      else if (bmi < 35)
35        System.out.println("You are seriously overweight");
36      else
37        System.out.println("You are gravely overweight");
38    }
39 }
```

```
Enter weight in pounds: 146 Enter
Enter height in inches: 70 Enter
Your BMI is 20.948603801493316
You are normal weight
```





line#	weight	height	WeightInKilograms	heightInMeters	bmi	output
9	146					
13		70				
19			66.22448602			
20				1.778		
21					20.9486	
25						Your BMI is 20.95
31						You are normal weight

Two constants `KILOGRAMS_PER_POUND` and `METERS_PER_INCH` are defined in lines 15–16. Using constants here makes programs easy to read.

3.11 Problem: Computing Taxes



Video Note
Use multiple alternative if statements

The United States federal personal income tax is calculated based on filing status and taxable income. There are four filing statuses: single filers, married filing jointly, married filing separately, and head of household. The tax rates vary every year. Table 3.2 shows the rates for 2009. If you are, say, single with a taxable income of $10,000, the first $8,350 is taxed at 10% and the other $1,650 is taxed at 15%. So, your tax is $1,082.5

TABLE 3.2 2009 U.S. Federal Personal Tax Rates

Marginal Tax Rate	*Single*	*Married Filing Jointly or Qualified Widow(er)*	*Married Filing Separately*	*Head of Household*
10%	$0 – $8,350	$0 – $16,700	$0 – $8,350	$0 – $11,950
15%	$8,351 – $33,950	$16,701 – $67,900	$8,351 – $33,950	$11,951 – $45,500
25%	$33,951 – $82,250	$67,901 – $137,050	$33,951 – $68,525	$45,501 – $117,450
28%	$82,251 – $171,550	$137,051 – $208,850	$68,525 – $104,425	$117,451 – $190,200
33%	$171,551 – $372,950	$208,851 – $372,950	$104,426 – $186,475	$190,201 – $372,950
35%	$372,951+	$372,951+	$186,476+	$372,951+

You are to write a program to compute personal income tax. Your program should prompt the user to enter the filing status and taxable income and compute the tax. Enter **0** for single filers, **1** for married filing jointly, **2** for married filing separately, and **3** for head of household.

Your program computes the tax for the taxable income based on the filing status. The filing status can be determined using **if** statements outlined as follows:

```
if (status == 0) {
  // Compute tax for single filers
}
else if (status == 1) {
  // Compute tax for married filing jointly
}
else if (status == 2) {
  // Compute tax for married filing separately
}
else if (status == 3) {
  // Compute tax for head of household
}
else {
  // Display wrong status
}
```

For each filing status there are six tax rates. Each rate is applied to a certain amount of taxable income. For example, of a taxable income of $400,000 for single filers, $8,350 is taxed at 10%, (33,950 − 8,350) at 15%, (82,250 − 33,950) at 25%, (171,550 − 82,250) at 28%, (372,950 − 171,550) at 33%, and (400,000 − 372,950) at 35%.

Listing 3.6 gives the solution to compute taxes for single filers. The complete solution is left as an exercise.

LISTING 3.6 ComputeTax.java

```
1  import java.util.Scanner;
2
3  public class ComputeTax {
4    public static void main(String[] args) {
5      // Create a Scanner
6      Scanner input = new Scanner(System.in);
7
8      // Prompt the user to enter filing status
9      System.out.print(
10       "(0-single filer, 1-married jointly,\n" +
11       "2-married separately, 3-head of household)\n" +
12       "Enter the filing status: ");
13     int status = input.nextInt();                          input status
14
15     // Prompt the user to enter taxable income
16     System.out.print("Enter the taxable income: ");
17     double income = input.nextDouble();                    input income
18
19     // Compute tax
20     double tax = 0;
21
22     if (status == 0) { // Compute tax for single filers    compute tax
23       if (income <= 8350)
24         tax = income * 0.10;
25       else if (income <= 33950)
26         tax = 8350 * 0.10 + (income - 8350) * 0.15;
27       else if (income <= 82250)
28         tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
```

```
29            (income - 33950) * 0.25;
30        else if (income <= 171550)
31          tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
32            (82250 - 33950) * 0.25 + (income - 82250) * 0.28;
33        else if (income <= 372950)
34          tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
35            (82250 - 33950) * 0.25 + (171550 - 82250) * 0.28 +
36            (income - 171550) * 0.35;
37        else
38          tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
39            (82250 - 33950) * 0.25 + (171550 - 82250) * 0.28 +
40            (372950 - 171550) * 0.33 + (income - 372950) * 0.35;
41      }
42      else if (status == 1) {  // Compute tax for married file jointly
43        // Left as exercise
44      }
45      else if (status == 2) {  // Compute tax for married separately
46        // Left as exercise
47      }
48      else if (status == 3) {  // Compute tax for head of household
49        // Left as exercise
50      }
51      else {
52        System.out.println("Error: invalid status");
53        System.exit(0);                                          exit program
54      }
55
56      // Display the result
57      System.out.println("Tax is " + (int)(tax * 100) / 100.0);  display output
58    }
59 }
```

```
(0-single filer, 1-married jointly,
2-married separately, 3-head of household)
Enter the filing status: 0 Enter
Enter the taxable income: 400000 Enter
Tax is 117683.5
```





line#	status	income	tax	output
13	0			
17		400000		
20			0	
38			117683.5	
57				Tax is 117683.5

The program receives the filing status and taxable income. The multiple alternative **if** statements (lines 22, 42, 45, 48, 51) check the filing status and compute the tax based on the filing status.

System.exit(0) (line 53) is defined in the **System** class. Invoking this method terminates the program. The argument **0** indicates that the program is terminated normally. **System.exit(0)**

An initial value of **0** is assigned to **tax** (line 20). A syntax error would occur if it had no initial value, because all of the other statements that assign values to **tax** are within the **if**

statement. The compiler thinks that these statements may not be executed and therefore reports a syntax error.

test all cases

To test a program, you should provide the input that covers all cases. For this program, your input should cover all statuses (0, 1, 2, 3). For each status, test the tax for each of the six brackets. So, there are a total of 24 cases.



Tip

incremental development and testing

For all programs, you should write a small amount of code and test it before moving on to add more code. This is called *incremental development and testing*. This approach makes debugging easier, because the errors are likely in the new code you just added.

3.12 Logical Operators

Sometimes, whether a statement is executed is determined by a combination of several conditions. You can use logical operators to combine these conditions. *Logical operators*, also known as *Boolean operators*, operate on Boolean values to create a new Boolean value. Table 3.3 gives a list of Boolean operators. Table 3.4 defines the not (`!`) operator. The not (`!`) operator negates `true` to `false` and `false` to `true`. Table 3.5 defines the and (`&&`) operator. The and (`&&`) of two Boolean operands is `true` if and only if both operands are `true`. Table 3.6 defines the or (`||`) operator. The or (`||`) of two Boolean operands is `true` if at least one of the operands is `true`. Table 3.7 defines the exclusive or (`^`) operator. The exclusive or (`^`) of two Boolean operands is `true` if and only if the two operands have different Boolean values.

TABLE 3.3 Boolean Operators

Operator	*Name*	*Description*
`!`	not	logical negation
`&&`	and	logical conjunction
`||`	or	logical disjunction
`^`	exclusive or	logical exclusion

TABLE 3.4 Truth Table for Operator `!`

p	*!p*	*Example (assume age = 24, gender = 'F')*
`true`	`false`	`!(age > 18)` is `false`, because `(age > 18)` is `true`.
`false`	`true`	`!(gender == 'M')` is `true`, because `(gender == 'M')` is `false`.

TABLE 3.5 Truth Table for Operator `&&`

p1	*p2*	*p1 && p2*	*Example (assume* age = 24, gender = 'F')
`false`	`false`	`false`	`(age > 18) && (gender == 'F')` is `true`, because `(age > 18)` and `(gender == 'F')` are both `true`.
`false`	`true`	`false`	
`true`	`false`	`false`	`(age > 18) && (gender != 'F')` is `false`, because `(gender != 'F')` is `false`.
`true`	`true`	`true`	

TABLE 3.6 Truth Table for Operator ||

p1	*p2*	*p1* || *p2*	*Example (assume* age = 24, gender = 'F'*)*
`false`	`false`	`false`	`(age > 34) || (gender == 'F')` is `true`, because `(gender == 'F')` is `true`.
`false`	`true`	`true`	
`true`	`false`	`true`	`(age > 34) || (gender == 'M')` is `false`, because `(age > 34)` and `(gender == 'M')` are both `false`.
`true`	`true`	`true`	

TABLE 3.7 Truth Table for Operator ^

p1	*p2*	*p1* ^ *p2*	*Example (assume* age = 24, gender = 'F'*)*
`false`	`false`	`false`	`(age > 34) ^ (gender == 'F')` is `true`, because `(age > 34)` is `false` but `(gender == 'F')` is `true`.
`false`	`true`	`true`	
`true`	`false`	`true`	`(age > 34) || (gender == 'M')` is `false`, because `(age > 34)` and `(gender == 'M')` are both `false`.
`true`	`true`	`false`	

Listing 3.7 gives a program that checks whether a number is divisible by 2 and 3, by 2 or 3, and by 2 or 3 but not both:

LISTING 3.7 TestBooleanOperators.java

```
1  import java.util.Scanner;                                          import class
2
3  public class TestBooleanOperators {
4    public static void main(String[] args) {
5      // Create a Scanner
6      Scanner input = new Scanner(System.in);
7
8      // Receive an input
9      System.out.print("Enter an integer: ");
10     int number = input.nextInt();                                  input
11
12     System.out.println("Is " + number +
13       "\n\tdivisible by 2 and 3? " +
14       (number % 2 == 0 && number % 3 == 0 )                        and
15       + "\n\tdivisible by 2 or 3? " +
16       (number % 2 == 0 || number % 3 == 0 ) +                      or
17       "\n\tdivisible by 2 or 3, but not both? "
18       + (number % 2 == 0 ^ number % 3 == 0 ));                     exclusive or
19   }
20 }
```

```
Enter an integer: 18 ↵Enter
Is 18
  divisible by 2 and 3? true
  divisible by 2 or 3? true
  divisible by 2 or 3, but not both? false
```



A long string is formed by concatenating the substrings in lines 12–18. The three `\n` characters display the string in four lines. `(number % 2 == 0 && number % 3 == 0)` (line 14) checks whether the number is divisible by `2` and `3`. `(number % 2 == 0 || number % 3 == 0)` (line 16) checks whether the number is divisible by `2` or `3`. `(number % 2 == 0 ^ number % 3 == 0)` (line 20) checks whether the number is divisible by `2` or `3`, but not both.



Caution

In mathematics, the expression

```
1 <= numberOfDaysInAMonth <= 31
```

is correct. However, it is incorrect in Java, because `1 <= numberOfDaysInAMonth` is evaluated to a `boolean` value, which cannot be compared with `31`. Here, two operands (a `boolean` value and a numeric value) are *incompatible*. The correct expression in Java is

incompatible operands

```
(1 <= numberOfDaysInAMonth) && (numberOfDaysInAMonth <= 31)
```



Note

As shown in the preceding chapter, a `char` value can be cast into an `int` value, and vice versa. A `boolean` value, however, cannot be cast into a value of another type, nor can a value of another type be cast into a `boolean` value.

cannot cast **`boolean`**



Note

De Morgan's law

De Morgan's law, named after Indian-born British mathematician and logician Augustus De Morgan (1806–1871), can be used to simplify Boolean expressions. The law states

```
!(condition1 && condition2) is same as !condition1 || !condition2
!(condition1 || condition2) is same as !condition1 && !condition2
```

For example,

```
!(n == 2 || n == 3) is same as n != 2 && n != 3
!(n % 2 == 0 && n % 3 == 0) is same as n % 2 != 0 || n % 3 != 0
```

If one of the operands of an `&&` operator is `false`, the expression is `false`; if one of the operands of an `||` operator is `true`, the expression is `true`. Java uses these properties to improve the performance of these operators. When evaluating `p1 && p2`, Java first evaluates `p1` and then, if `p1` is `true`, evaluates `p2`; if `p1` is `false`, it does not evaluate `p2`. When evaluating `p1 || p2`, Java first evaluates `p1` and then, if `p1` is `false`, evaluates `p2`; if `p1` is `true`, it does not evaluate `p2`. Therefore, `&&` is referred to as the *conditional* or *short-circuit AND* operator, and || is referred to as the *conditional* or *short-circuit OR* operator.

conditional operator
short-circuit operator

3.13 Problem: Determining Leap Year

leap year

A year is a *leap year* if it is divisible by `4` but not by `100` or if it is divisible by `400`. So you can use the following Boolean expressions to check whether a year is a leap year:

```
// A leap year is divisible by 4
boolean isLeapYear = (year % 4 == 0);

// A leap year is divisible by 4 but not by 100
isLeapYear = isLeapYear && (year % 100 != 0);

// A leap year is divisible by 4 but not by 100 or divisible by 400
isLeapYear = isLeapYear || (year % 400 == 0);
```

or you can combine all these expressions into one like this:

```
isLeapYear = (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
```

Listing 3.8 gives the program that lets the user enter a year and checks whether it is a leap year.

LISTING 3.8 LeapYear.java

```
1 import java.util.Scanner;
2
3 public class LeapYear {
4   public static void main(String[] args) {
5     // Create a Scanner
6     Scanner input = new Scanner(System.in);
7     System.out.print("Enter a year: ");
8     int year = input.nextInt();                                    input
9
10    // Check if the year is a leap year
11    boolean isLeapYear =                                           leap year?
12      (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
13
14    // Display the result
15    System.out.println(year + " is a leap year? " + isLeapYear);   display result
16  }
17 }
```

```
Enter a year: 2008 Enter
2008 is a leap year? true

Enter a year: 2002 Enter
2002 is a leap year? false
```



3.14 Problem: Lottery

Suppose you want to develop a program to play lottery. The program randomly generates a lottery of a two-digit number, prompts the user to enter a two-digit number, and determines whether the user wins according to the following rule:

1. If the user input matches the lottery in exact order, the award is $10,000.
2. If all the digits in the user input match all the digits in the lottery, the award is $3,000.
3. If one digit in the user input matches a digit in the lottery, the award is $1,000.

The complete program is shown in Listing 3.9.

LISTING 3.9 Lottery.java

```
1 import java.util.Scanner;
2
3 public class Lottery {
4   public static void main(String[] args) {
5     // Generate a lottery
6     int lottery = (int)(Math.random() * 100);                      generate a lottery
7
8     // Prompt the user to enter a guess
9     Scanner input = new Scanner(System.in);
10    System.out.print("Enter your lottery pick (two digits): ");
11    int guess = input.nextInt();                                   enter a guess
12
13    //   Get digits from lottery
14    int  lotteryDigit1 = lottery / 10;
```

```
15     int lotteryDigit2 = lottery % 10;
16
17     // Get digits from guess
18     int guessDigit1 = guess / 10;
19     int guessDigit2 = guess % 10;
20
21     System.out.println("The lottery number is " + lottery);
22
23     // Check the guess
24     if (guess == lottery)
25       System.out.println("Exact match: you win $10,000");
26     else if (guessDigit2 == lotteryDigit1
27           && guessDigit1 == lotteryDigit2)
28       System.out.println("Match all digits: you win $3,000");
29     else if (guessDigit1 == lotteryDigit1
30           || guessDigit1 == lotteryDigit2
31           || guessDigit2 == lotteryDigit1
32           || guessDigit2 == lotteryDigit2)
33       System.out.println("Match one digit: you win $1,000");
34     else
35       System.out.println("Sorry, no match");
36   }
37 }
```

exact match?

match all digits?

match one digit?



```
Enter your lottery pick (two digits): 45 Enter
The lottery number is 12
Sorry, no match
```



```
Enter your lottery pick: 23 Enter
The lottery number is 34
Match one digit: you win $1,000
```



line# / variable	6	11	14	15	18	19	33
lottery	34						
guess		23					
lotteryDigit1			3				
lotteryDigit2				4			
guessDigit1					2		
guessDigit2						3	
output							Match one digit: you win $1,000

The program generates a lottery using the `random()` method (line 6) and prompts the user to enter a guess (line 11). Note that `guess % 10` obtains the last digit from `guess` and `guess / 10` obtains the first digit from `guess`, since `guess` is a two-digit number (lines 18–19).

The program checks the guess against the lottery number in this order:

1. First check whether the guess matches the lottery exactly (line 24).
2. If not, check whether the reversal of the guess matches the lottery (lines 26–27).

3. If not, check whether one digit is in the lottery (lines 29–32).

4. If not, nothing matches.

3.15 switch Statements

The `if` statement in Listing 3.6, ComputeTax.java, makes selections based on a single `true` or `false` condition. There are four cases for computing taxes, which depend on the value of `status`. To fully account for all the cases, nested `if` statements were used. Overuse of nested `if` statements makes a program difficult to read. Java provides a `switch` statement to handle multiple conditions efficiently. You could write the following `switch` statement to replace the nested `if` statement in Listing 3.6:

```
switch (status) {
  case 0:  compute taxes for single filers;
           break;
  case 1:  compute taxes for married filing jointly;
           break;
  case 2:  compute taxes for married filing separately;
           break;
  case 3:  compute taxes for head of household;
           break;
  default: System.out.println("Errors: invalid status");
           System.exit(0);
}
```

The flow chart of the preceding `switch` statement is shown in Figure 3.5.

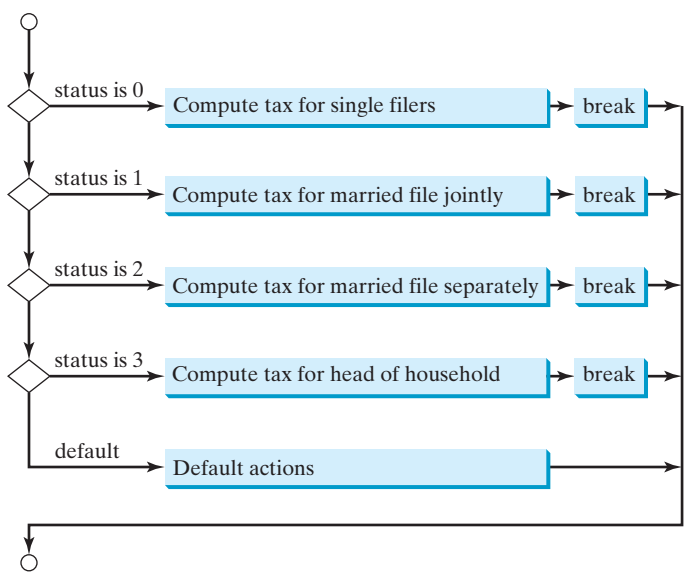


FIGURE 3.5 The `switch` statement checks all cases and executes the statements in the matched case.

This statement checks to see whether the status matches the value `0`, `1`, `2`, or `3`, in that order. If matched, the corresponding tax is computed; if not matched, a message is displayed. Here is the full syntax for the `switch` statement:

switch statement

```
switch (switch-expression) {
  case value1: statement(s)1;
               break;
```

```
    case value2: statement(s)2;
                 break;
    ...
    case valueN: statement(s)N;
                 break;
    default:     statement(s)-for-default;
}
```

The `switch` statement observes the following rules:

- The `switch-expression` must yield a value of `char`, `byte`, `short`, or `int` type and must always be enclosed in parentheses.
- The `value1`, ..., and `valueN` must have the same data type as the value of the `switch-expression`. Note that `value1`, ..., and `valueN` are constant expressions, meaning that they cannot contain variables, such as `1 + x`.
- When the value in a `case` statement matches the value of the `switch-expression`, the statements *starting from this case* are executed until either a `break` statement or the end of the switch statement is reached.
- The keyword `break` is optional. The `break` statement immediately ends the `switch` statement.
- The `default` case, which is optional, can be used to perform actions when none of the specified cases matches the `switch-expression`.
- The `case` statements are checked in sequential order, but the order of the cases (including the default case) does not matter. However, it is good programming style to follow the logical sequence of the cases and place the default case at the end.



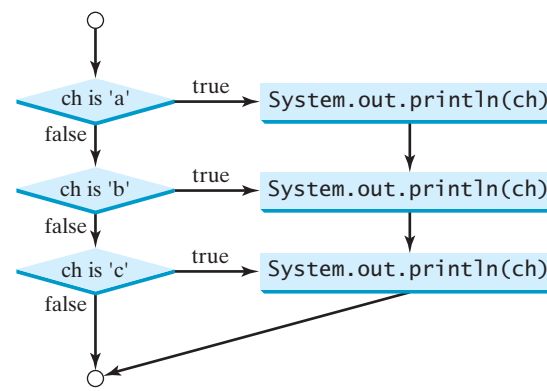
Caution

without **break**

fall-through behavior

Do not forget to use a `break` statement when one is needed. Once a case is matched, the statements starting from the matched case are executed until a `break` statement or the end of the `switch` statement is reached. This is referred to as *fall-through* behavior. For example, the following code prints character `a` three times if `ch` is `'a'`:

```
switch (ch) {
  case 'a': System.out.println(ch);
  case 'b': System.out.println(ch);
  case 'c': System.out.println(ch);
}
```





Tip

To avoid programming errors and improve code maintainability, it is a good idea to put a comment in a case clause if `break` is purposely omitted.

3.16 Conditional Expressions

You might want to assign a value to a variable that is restricted by certain conditions. For example, the following statement assigns `1` to `y` if `x` is greater than `0`, and `-1` to `y` if `x` is less than or equal to `0`.

```
if (x > 0)
  y = 1;
else
  y = -1;
```

Alternatively, as in this example, you can use a conditional expression to achieve the same result.

```
y = (x > 0) ? 1 : -1;
```

Conditional expressions are in a completely different style, with no explicit `if` in the statement. The syntax is shown below:

```
boolean-expression ? expression1 : expression2;
```

conditional expression

The result of this conditional expression is `expression1` if `boolean-expression` is true; otherwise the result is `expression2`.

Suppose you want to assign the larger number between variable `num1` and `num2` to `max`. You can simply write a statement using the conditional expression:

```
max = (num1 > num2) ? num1 : num2;
```

For another example, the following statement displays the message "num is even" if `num` is even, and otherwise displays "num is odd."

```
System.out.println((num % 2 == 0) ? "num is even" : "num is odd");
```



Note

The symbols `?` and `:` appear together in a conditional expression. They form a conditional operator. It is called a *ternary operator* because it uses three operands. It is the only ternary operator in Java.

3.17 Formatting Console Output

If you wish to display only two digits after the decimal point in a floating-point value, you may write the code like this:

```
double x = 2.0 / 3;
System.out.println("x is " + (int)(x * 100) / 100.0);
```

```
x is 0.66
```



However, a better way to accomplish this task is to format the output using the `printf` method. The syntax to invoke this method is

`printf`

```
System.out.printf(format, item1, item2, ..., itemk)
```

where `format` is a string that may consist of substrings and format specifiers.

specifier

A format specifier specifies how an item should be displayed. An item may be a numeric value, a character, a Boolean value, or a string. A simple specifier consists of a percent sign (%) followed by a conversion code. Table 3.8 lists some frequently used simple specifiers:

TABLE 3.8 Frequently Used Specifiers

Specifier	*Output*	*Example*
`%b`	a Boolean value	true or false
`%c`	a character	'a'
`%d`	a decimal integer	200
`%f`	a floating-point number	45.460000
`%e`	a number in standard scientific notation	4.556000e+01
`%s`	a string	"Java is cool"

Here is an example:

```
int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);
```

items

```
display              count is 5 and amount is 45.560000
```

Items must match the specifiers in order, in number, and in exact type. For example, the specifier for `count` is `%d` and for `amount` is `%f`. By default, a floating-point value is displayed with six digits after the decimal point. You can specify the width and precision in a specifier, as shown in the examples in Table 3.9.

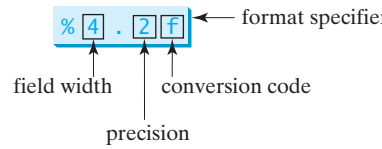
TABLE 3.9 Examples of Specifying Width and Precision

Example	*Output*
`%5c`	Output the character and add four spaces before the character item.
`%6b`	Output the Boolean value and add one space before the false value and two spaces before the true value.
`%5d`	Output the integer item with width at least 5. If the number of digits in the item is $<$ 5, add spaces before the number. If the number of digits in the item is $>$ 5, the width is automatically increased.
`%10.2f`	Output the floating-point item with width at least 10 including a decimal point and two digits after the point. Thus there are 7 digits allocated before the decimal point. If the number of digits before the decimal point in the item is $<$ 7, add spaces before the number. If the number of digits before the decimal point in the item is $>$ 7, the width is automatically increased.
`%10.2e`	Output the floating-point item with width at least 10 including a decimal point, two digits after the point and the exponent part. If the displayed number in scientific notation has width less than 10, add spaces before the number.
`%12s`	Output the string with width at least 12 characters. If the string item has less than 12 characters, add spaces before the string. If the string item has more than 12 characters, the width is automatically increased.

The code presented in the beginning of this section for displaying only two digits after the decimal point in a floating-point value can be revised using the **printf** method as follows:

```
double x = 2.0 / 3;
System.out.printf("x is %4.2f", x);
```

```
display            x is 0.67
```

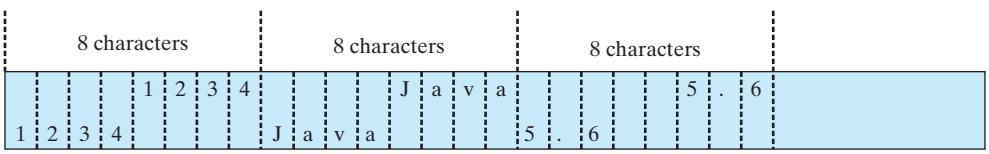


By default, the output is right justified. You can put the minus sign (-) in the specifier to specify that the item is left justified in the output within the specified field. For example, the following statements

left justify

```
System.out.printf("%8d%8s%8.1f\n", 1234, "Java", 5.6);
System.out.printf("%-8d%-8s%-8.1f \n", 1234, "Java", 5.6);
```

display





Caution

The items must match the specifiers in exact type. The item for the specifier **%f** or **%e** must be a floating-point type value such as 40.0, not 40. Thus an **int** variable cannot match **%f** or **%e**.



Tip

The **%** sign denotes a specifier. To output a literal **%** in the format string, use **%%**.

3.18 Operator Precedence and Associativity

Operator precedence and associativity determine the order in which operators are evaluated. Suppose that you have this expression:

```
3 + 4 * 4 > 5 * (4 + 3) - 1
```

What is its value? What is the execution order of the operators?

Arithmetically, the expression in the parentheses is evaluated first. (Parentheses can be nested, in which case the expression in the inner parentheses is executed first.) When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the associativity rule.

The precedence rule defines precedence for operators, as shown in Table 3.10, which contains the operators you have learned so far. Operators are listed in decreasing order of precedence from top to bottom. Operators with the same precedence appear in the same group. (See Appendix C, "Operator Precedence Chart," for a complete list of Java operators and their precedence.)

precedence

If operators with the same precedence are next to each other, their *associativity* determines the order of evaluation. All binary operators except assignment operators are *left associative*. For example, since + and - are of the same precedence and are left associative, the expression

associativity

```
a - b + c - d   ——equivalent——   ((a - b) + c) - d
```

TABLE 3.10 Operator Precedence Chart

Precedence	*Operator*
↓	`var++` and `var--` (Postfix)
	`+`, `-` (Unary plus and minus), `++var` and `--var` (Prefix)
	(type) (Casting)
	`!` (Not)
	`*`, `/`, `%` (Multiplication, division, and remainder)
	`+`, `-` (Binary addition and subtraction)
	`<`, `<=`, `>`, `>=` (Comparison)
	`==`, `!=` (Equality)
	`^` (Exclusive OR)
	`&&` (AND)
	`||` (OR)
	`=`, `+=`, `-=`, `*=`, `/=`, `%=` (Assignment operator)

Assignment operators are *right associative*. Therefore, the expression

```
a = b += c = 5   ——equivalent——   a = (b += (c = 5))
```

Suppose `a`, `b`, and `c` are 1 before the assignment; after the whole expression is evaluated, `a` becomes 6, `b` becomes 6, and `c` becomes 5. Note that left associativity for the assignment operator would not make sense.



Note

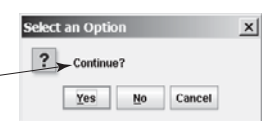
Java has its own way to evaluate an expression internally. The result of a Java evaluation is the same as that of its corresponding arithmetic evaluation. Interested readers may refer to Supplement III.B for more discussions on how an expression is evaluated in Java *behind the scenes*.

behind the scenes

3.19 (GUI) Confirmation Dialogs

You have used `showMessageDialog` to display a message dialog box and `showInputDialog` to display an input dialog box. Occasionally it is useful to answer a question with a confirmation dialog box. A confirmation dialog can be created using the following statement:

```
int option =
  JOptionPane.showConfirmDialog
    (null, "Continue");
```



When a button is clicked, the method returns an option value. The value is `JOptionPane.YES_OPTION` (`0`) for the *Yes* button, `JOptionPane.NO_OPTION` (`1`) for the *No* button, and `JOptionPane.CANCEL_OPTION` (`2`) for the *Cancel* button.

You may rewrite the guess-birthday program in Listing 3.3 using confirmation dialog boxes, as shown in Listing 3.10. Figure 3.6 shows a sample run of the program for the day `19`.

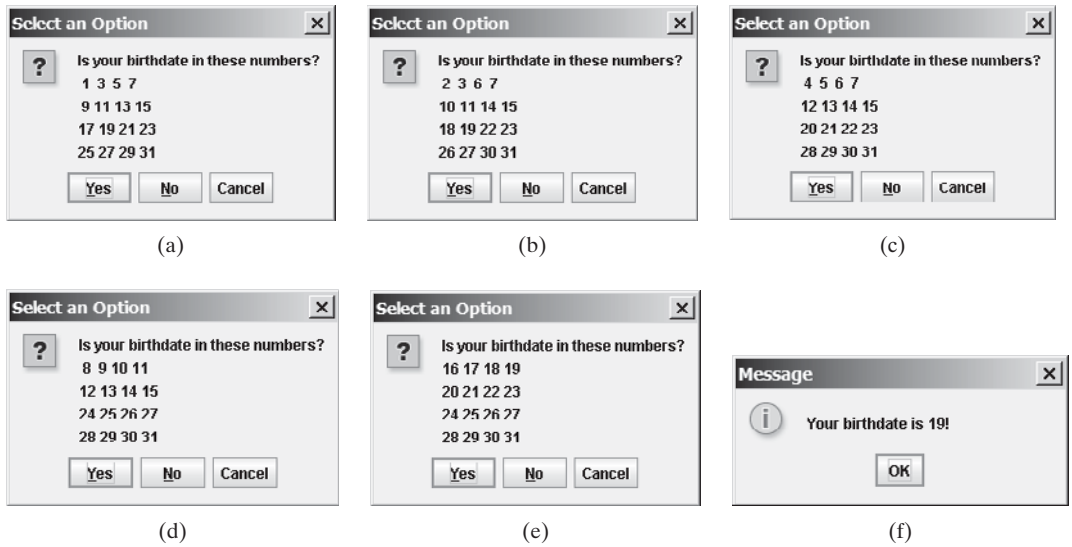


FIGURE 3.6 Click Yes in (a), Yes in (b), No in (c), No in (d), and Yes in (e).

LISTING 3.10 GuessBirthdayUsingConfirmationDialog.java

```
1  import javax.swing.JOptionPane;                                    import class
2
3  public class GuessBirthdayUsingConfirmationDialog {
4    public static void main(String[] args) {
5      String set1 =                                                  set1
6        " 1  3  5  7\n" +
7        " 9 11 13 15\n" +
8        "17 19 21 23\n" +
9        "25 27 29 31";
10
11     String set2 =                                                  set2
12       " 2  3  6  7\n" +
13       "10 11 14 15\n" +
14       "18 19 22 23\n" +
15       "26 27 30 31";
16
17     String set3 =                                                  set3
18       " 4  5  6  7\n" +
19       "12 13 14 15\n" +
20       "20 21 22 23\n" +
21       "28 29 30 31";
22
23     String set4 =                                                  set4
24       " 8  9 10 11\n" +
25       "12 13 14 15\n" +
26       "24 25 26 27\n" +
27       "28 29 30 31";
28
29     String set5 =                                                  set5
30       "16 17 18 19\n" +
31       "20 21 22 23\n" +
32       "24 25 26 27\n" +
33       "28 29 30 31";
34
```

confirmation dialog

in set1?

in set2?

in set3?

in set4?

in set5?

```
35      int day = 0;
36
37      // Prompt the user to answer questions
38      int answer = JOptionPane.showConfirmDialog(null,
39        "Is your birthday in these numbers?\n" + set1);
40
41      if (answer == JOptionPane.YES_OPTION)
42        day += 1;
43
44      answer = JOptionPane.showConfirmDialog(null,
45        "Is your birthday in these numbers?\n" + set2);
46
47      if (answer == JOptionPane.YES_OPTION)
48        day += 2;
49
50      answer = JOptionPane.showConfirmDialog(null,
51        "Is your birthday in these numbers?\n" + set3);
52
53      if (answer == JOptionPane.YES_OPTION)
54        day += 4;
55
56      answer = JOptionPane.showConfirmDialog(null,
57        "Is your birthday in these numbers?\n" + set4);
58
59      if (answer == JOptionPane.YES_OPTION)
60        day += 8;
61
62      answer = JOptionPane.showConfirmDialog(null,
63        "Is your birthday in these numbers?\n" + set5);
64
65      if (answer == JOptionPane.YES_OPTION)
66        day += 16;
67
68      JOptionPane.showMessageDialog(null, "Your birthday is " +
69        day + "!");
70    }
71 }
```

The program displays confirmation dialog boxes to prompt the user to answer whether a number is in Set1 (line 38), Set2 (line 44), Set3 (line 50), Set4 (line 56), and Set5 (line 62). If the answer is Yes, the first number in the set is added to `day` (lines 42, 48, 54, 60, and 66).

Key Terms

Boolean expression 72
Boolean value 72
`boolean` type 72
`break` statement 94
conditional operator 90
dangling-`else` ambiguity 82
fall-through behavior 94
operator associativity 97
operator precedence 97
selection statement 74
short-circuit evaluation 90

Chapter Summary

1. A `boolean` variable stores a `true` or `false` value.
2. The relational operators (`<`, `<=`, `==`, `!=`, `>`, `>=`) work with numbers and characters, and yield a Boolean value.

3. The Boolean operators `&&`, `||`, `!`, and `^` operate with Boolean values and variables.

4. When evaluating `p1 && p2`, Java first evaluates `p1` and then evaluates `p2` if `p1` is `true`; if `p1` is `false`, it does not evaluate `p2`. When evaluating `p1 || p2`, Java first evaluates `p1` and then evaluates `p2` if `p1` is `false`; if `p1` is `true`, it does not evaluate `p2`. Therefore, `&&` is referred to as the conditional or short-circuit AND operator, and `||` is referred to as the conditional or short-circuit OR operator.

5. Selection statements are used for programming with alternative courses. There are several types of selection statements: `if` statements, `if ... else` statements, nested `if` statements, `switch` statements, and conditional expressions.

6. The various `if` statements all make control decisions based on a Boolean expression. Based on the `true` or `false` evaluation of the expression, these statements take one of two possible courses.

7. The `switch` statement makes control decisions based on a switch expression of type `char`, `byte`, `short`, or `int`.

8. The keyword `break` is optional in a switch statement, but it is normally used at the end of each case in order to terminate the remainder of the `switch` statement. If the `break` statement is not present, the next `case` statement will be executed.

Review Questions

Section 3.2

3.1 List six comparison operators.

3.2 Can the following conversions involving casting be allowed? If so, find the converted result.

```
boolean b = true;
i = (int)b;
```

```
int i = 1;
boolean b = (boolean)i;
```

Sections 3.3–3.11

3.3 What is the printout of the code in (a) and (b) if `number` is `30` and `35`, respectively?

```
if (number % 2 == 0)
  System.out.println(number + " is even.");

  System.out.println(number + " is odd.");
```

(a)

```
if (number % 2 == 0)
  System.out.println(number + " is even.");
else
  System.out.println(number + " is odd.");
```

(b)

3.4 Suppose `x = 3` and `y = 2`; show the output, if any, of the following code. What is the output if `x = 3` and `y = 4`? What is the output if `x = 2` and `y = 2`? Draw a flow chart of the code:

```
if (x > 2) {
  if (y > 2) {
```

```
    z = x + y;
    System.out.println("z is " + z);
  }
}
else
  System.out.println("x is " + x);
```

3.5 Which of the following statements are equivalent? Which ones are correctly indented?

```
if (i > 0) if
(j > 0)
x = 0; else
if (k > 0) y = 0;
else z = 0;
```

(a)

```
if (i > 0) {
  if (j > 0)
    x = 0;
  else if (k > 0)
    y = 0;
}
else
  z = 0;
```

(b)

```
if (i > 0)
  if (j > 0)
    x = 0;
  else if (k > 0)
    y = 0;
  else
    z = 0;
```

(c)

```
if (i > 0)
  if (j > 0)
    x = 0;
  else if (k > 0)
    y = 0;
else
  z = 0;
```

(d)

3.6 Suppose `x = 2` and `y = 3`. Show the output, if any, of the following code. What is the output if `x = 3` and `y = 2`? What is the output if `x = 3` and `y = 3`?

(*Hint*: Indent the statement correctly first.)

```
if (x > 2)
  if (y > 2) {
    int z = x + y;
    System.out.println("z is " + z);
  }
else
  System.out.println("x is " + x);
```

3.7 Are the following two statements equivalent?

```
if (income <= 10000)
  tax = income * 0.1;
else if (income <= 20000)
  tax = 1000 +
    (income - 10000) * 0.15;
```

```
if (income <= 10000)
  tax = income * 0.1;
else if (income > 10000 &&
         income <= 20000)
  tax = 1000 +
    (income - 10000) * 0.15;
```

3.8 Which of the following is a possible output from invoking `Math.random()`?

`323.4`, `0.5`, `34`, `1.0`, `0.0`, `0.234`

3.9 How do you generate a random integer `i` such that $0 \le i < 20$? How do you generate a random integer `i` such that $10 \le i < 20$? How do you generate a random integer `i` such that $10 \le i \le 50$?

3.10 Write an `if` statement that assigns `1` to `x` if `y` is greater than `0`.

3.11 (a) Write an `if` statement that increases `pay` by 3% if `score` is greater than `90`. (b) Write an `if` statement that increases `pay` by 3% if `score` is greater than `90`, otherwise increases `pay` by 1%.

3.12 What is wrong in the following code?

```
if (score >= 60.0)
  grade = 'D';
else if (score >= 70.0)
  grade = 'C';
else if (score >= 80.0)
  grade = 'B';
else if (score >= 90.0)
  grade = 'A';
else
  grade = 'F';
```

3.13 Rewrite the following statement using a Boolean expression:

```
if (count % 10 == 0)
  newLine = true;
else
  newLine = false;
```

Sections 3.12–3.14

3.14 Assuming that `x` is `1`, show the result of the following Boolean expressions.

```
(true) && (3 > 4)
!(x > 0) && (x > 0)
(x > 0) || (x < 0)
(x != 0) || (x == 0)
(x >= 0) || (x < 0)
(x != 1) == !(x == 1)
```

3.15 Write a Boolean expression that evaluates to `true` if a number stored in variable `num` is between `1` and `100`.

3.16 Write a Boolean expression that evaluates to `true` if a number stored in variable `num` is between `1` and `100` or the number is negative.

3.17 Assume that `x` and `y` are `int` type. Which of the following are legal Java expressions?

```
x > y > 0
x = y && y
x /= y
x or y
x and y
(x != 0) || (x = 0)
```

3.18 Suppose that `x` is `1`. What is `x` after the evaluation of the following expression?

```
(x >= 1) && (x++ > 1)
(x > 1) && (x++ > 1)
```

3.19 What is the value of the expression `ch >= 'A' && ch <= 'Z'` if `ch` is `'A'`, `'p'`, `'E'`, or `'5'`?

3.20 Suppose, when you run the program, you enter input `2 3 6` from the console. What is the output?

```
public class Test {
  public static void main(String[] args) {
    java.util.Scanner input = new java.util.Scanner(System.in);
    double x = input.nextDouble();
```

```
    double y = input.nextDouble();
    double z = input.nextDouble();

    System.out.println("(x < y && y < z) is " + (x < y && y < z));
    System.out.println("(x < y || y < z) is " + (x < y || y < z));
    System.out.println("!(x < y) is " + !(x < y));
    System.out.println("(x + y < z) is " + (x + y < z));
    System.out.println("(x + y < z) is " + (x + y < z));
  }
}
```

3.21 Write a Boolean expression that evaluates `true` if `age` is greater than `13` and less than `18`.

3.22 Write a Boolean expression that evaluates `true` if `weight` is greater than `50` or height is greater than `160`.

3.23 Write a Boolean expression that evaluates `true` if `weight` is greater than `50` and height is greater than `160`.

3.24 Write a Boolean expression that evaluates `true` if either `weight` is greater than `50` or height is greater than `160`, but not both.

Section 3.15

3.25 What data types are required for a `switch` variable? If the keyword `break` is not used after a case is processed, what is the next statement to be executed? Can you convert a `switch` statement to an equivalent `if` statement, or vice versa? What are the advantages of using a `switch` statement?

3.26 What is `y` after the following `switch` statement is executed?

```
x = 3; y = 3;
switch (x + 3) {
  case 6:  y = 1;
  default: y += 1;
}
```

3.27 Use a `switch` statement to rewrite the following `if` statement and draw the flow chart for the `switch` statement:

```
if (a == 1)
  x += 5;
else if (a == 2)
  x += 10;
else if (a == 3)
  x += 16;
else if (a == 4)
  x += 34;
```

3.28 Write a `switch` statement that assigns a `String` variable `dayName` with Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, if `day` is `0`, `1`, `2`, `3`, `4`, `5`, `6`, accordingly.

Section 3.16

3.29 Rewrite the following `if` statement using the conditional operator:

```
if (count % 10 == 0)
  System.out.print(count + "\n");
else
  System.out.print(count + " ");
```

3.30 Rewrite the following statement using a conditional expression:

```
if (temperature > 90)
  pay = pay * 1.5;
else
  pay = pay * 1.1;
```

Section 3.17

3.31 What are the specifiers for outputting a Boolean value, a character, a decimal integer, a floating-point number, and a string?

3.32 What is wrong in the following statements?

```
(a) System.out.printf("%5d %d", 1, 2, 3);
(b) System.out.printf("%5d %f", 1);
(c) System.out.printf("%5d %f", 1, 2);
```

3.33 Show the output of the following statements.

```
(a) System.out.printf("amount is %f %e\n", 32.32, 32.32);
(b) System.out.printf("amount is %5.4f %5.4e\n", 32.32, 32.32);
(c) System.out.printf("%6b\n", (1 > 2));
(d) System.out.printf("%6s\n", "Java");
(e) System.out.printf("%-6b%s\n", (1 > 2), "Java");
(f) System.out.printf("%6b%-s\n", (1 > 2), "Java");
```

3.34 How do you create a formatted string?

Section 3.18

3.35 List the precedence order of the Boolean operators. Evaluate the following expressions:

```
true || true && false
true && true || false
```

3.36 True or false? All the binary operators except `=` are left associative.

3.37 Evaluate the following expressions:

```
2 * 2 - 3 > 2 && 4 - 2 > 5
2 * 2 - 3 > 2 || 4 - 2 > 5
```

3.38 Is `(x > 0 && x < 10)` the same as `((x > 0) && (x < 10))`? Is `(x > 0 || x < 10)` the same as `((x > 0) || (x < 10))`? Is `(x > 0 || x < 10 && y < 0)` the same as `(x > 0 || (x < 10 && y < 0))`?

Section 3.19

3.39 How do you display a confirmation dialog? What value is returned when invoking `JOptionPane.showConfirmDialog`?

Programming Exercises



Pedagogical Note

For each exercise, students should carefully analyze the problem requirements and design strategies for solving the problem before coding.

think before coding

document analysis and design

Pedagogical Note

Instructors may ask students to document analysis and design for selected exercises. Students should use their own words to analyze the problem, including the input, output, and what needs to be computed, and describe how to solve the problem in pseudocode.

learn from mistakes

Debugging Tip

Before you ask for help, read and explain the program to yourself, and trace it using several representative inputs by hand or using an IDE debugger. You learn how to program by debugging your own mistakes.

Section 3.2

3.1* (*Algebra: solving quadratic equations*) The two roots of a quadratic equation $ax^2 + bx + c = 0$ can be obtained using the following formula:

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \text{ and } r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

$b^2 - 4ac$ is called the discriminant of the quadratic equation. If it is positive, the equation has two real roots. If it is zero, the equation has one root. If it is negative, the equation has no real roots.

Write a program that prompts the user to enter values for a, b, and c and displays the result based on the discriminant. If the discriminant is positive, display two roots. If the discriminant is 0, display one root. Otherwise, display "The equation has no real roots".

Note you can use `Math.pow(x, 0.5)` to compute $\sqrt{x}$. Here are some sample runs.



```
Enter a, b, c: 1.0 3 1 Enter
The roots are -0.381966 and -2.61803
```



```
Enter a, b, c: 1 2.0 1 Enter
The root is -1
```



```
Enter a, b, c: 1 2 3 Enter
The equation has no real roots
```

3.2 (*Checking whether a number is even*) Write a program that reads an integer and checks whether it is even. Here are the sample runs of this program:



```
Enter an integer: 25 Enter
Is 25 an even number? false
```



```
Enter an integer: 2000 Enter
Is 2000 an even number? true
```

Sections 3.3–3.8

3.3* (*Algebra: solving* 2×2 *linear equations*) You can use Cramer's rule to solve the following 2×2 system of linear equation:

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned} \qquad x = \frac{ed - bf}{ad - bc} \qquad y = \frac{af - ec}{ad - bc}$$

Write a program that prompts the user to enter **a**, **b**, **c**, **d**, **e**, and **f** and display the result. If $ad - bc$ is **0**, report that "The equation has no solution".

```
Enter a, b, c, d, e, f: 9.0 4.0 3.0 -5.0 -6.0 -21.0 Enter
x is -2.0 and y is 3.0
```



```
Enter a, b, c, d, e, f: 1.0 2.0 2.0 4.0 4.0 5.0 Enter
The equation has no solution
```



3.4** (*Game: learning addition*) Write a program that generates two integers under 100 and prompts the user to enter the sum of these two integers. The program then reports true if the answer is correct, false otherwise. The program is similar to Listing 3.1.

3.5** (*Game: addition for three numbers*) The program in Listing 3.1 generates two integers and prompts the user to enter the sum of these two integers. Revise the program to generate three single-digit integers and prompt the user to enter the sum of these three integers.

3.6* (*Health application: BMI*) Revise Listing 3.5, ComputeBMI.java, to let the user enter weight, feet, and inches. For example, if a person is **5** feet and **10** inches, you will enter **5** for feet and **10** for inches.

3.7 (*Financial application: monetary units*) Modify Listing 2.10, ComputeChange.java, to display the nonzero denominations only, using singular words for single units such as **1** dollar and **1** penny, and plural words for more than one unit such as **2** dollars and **3** pennies. (Use input **23.67** to test your program.)



Video Note
Sort three integers

3.8* (*Sorting three integers*) Write a program that sorts three integers. The integers are entered from the input dialogs and stored in variables **num1**, **num2**, and **num3**, respectively. The program sorts the numbers so that $num1 \le num2 \le num3$.

3.9 (*Business: checking ISBN*) An **ISBN** (International Standard Book Number) consists of 10 digits $d_1d_2d_3d_4d_5d_6d_7d_8d_9d_{10}$. The last digit d_{10} is a checksum, which is calculated from the other nine digits using the following formula:

$$(d_1 \times 1 + d_2 \times 2 + d_3 \times 3 + d_4 \times 4 + d_5 \times 5 + d_6 \times 6 + d_7 \times 7 + d_8 \times 8 + d_9 \times 9) \% 11$$

If the checksum is **10**, the last digit is denoted X according to the ISBN convention. Write a program that prompts the user to enter the first 9 digits and displays the 10-digit ISBN (including leading zeros). Your program should read the input as an integer. For example, if you enter 013601267, the program should display 0136012671.

3.10* (*Game: addition quiz*) Listing 3.4, SubtractionQuiz.java, randomly generates a subtraction question. Revise the program to randomly generate an addition question with two integers less than **100**.

Sections 3.9–3.19

3.11* (*Finding the number of days in a month*) Write a program that prompts the user to enter the month and year and displays the number of days in the month. For example, if the user entered month **2** and year **2000**, the program should display that February 2000 has 29 days. If the user entered month **3** and year **2005**, the program should display that March 2005 has 31 days.

3.12 (*Checking a number*) Write a program that prompts the user to enter an integer and checks whether the number is divisible by both **5** and **6**, or neither of them, or just one of them. Here are some sample runs for inputs **10**, **30**, and **23**.

```
10 is divisible by 5 or 6, but not both
30 is divisible by both 5 and 6
23 is not divisible by either 5 or 6
```

3.13 (*Financial application: computing taxes*) Listing 3.6, ComputeTax.java, gives the source code to compute taxes for single filers. Complete Listing 3.6 to give the complete source code.

3.14 (*Game: head or tail*) Write a program that lets the user guess the head or tail of a coin. The program randomly generates an integer **0** or **1**, which represents head or tail. The program prompts the user to enter a guess and reports whether the guess is correct or incorrect.

3.15* (*Game: lottery*) Revise Listing 3.9, Lottery.java, to generate a lottery of a three-digit number. The program prompts the user to enter a three-digit number and determines whether the user wins according to the following rule:

1. If the user input matches the lottery in exact order, the award is $10,000.
2. If all the digits in the user input match all the digits in the lottery, the award is $3,000.
3. If one digit in the user input matches a digit in the lottery, the award is $1,000.

3.16 (*Random character*) Write a program that displays a random uppercase letter using the `Math.random()` method.

3.17* (*Game: scissor, rock, paper*) Write a program that plays the popular scissor-rock-paper game. (A scissor can cut a paper, a rock can knock a scissor, and a paper can wrap a rock.) The program randomly generates a number **0**, **1**, or **2** representing scissor, rock, and paper. The program prompts the user to enter a number **0**, **1**, or **2** and displays a message indicating whether the user or the computer wins, loses, or draws. Here are sample runs:



```
scissor (0), rock (1), paper (2): 1 Enter
The computer is scissor. You are rock. You won
```



```
scissor (0), rock (1), paper (2): 2 Enter
The computer is paper. You are paper too. It is a draw
```

3.18* (*Using the input dialog box*) Rewrite Listing 3.8, LeapYear.java, using the input dialog box.

3.19 (*Validating triangles*) Write a program that reads three edges for a triangle and determines whether the input is valid. The input is valid if the sum of any two edges is greater than the third edge. Here are the sample runs of this program:

```
Enter three edges: 1 2.5 1 Enter
Can edges 1, 2.5, and 1 form a triangle? false
```



```
Enter three edges: 2.5 2 1 Enter
Can edges 2.5, 2, and 1 form a triangle? true
```



3.20* (*Science: wind-chill temperature*) Exercise 2.17 gives a formula to compute the wind-chill temperature. The formula is valid for temperature in the range between −58°F and 41°F and wind speed greater than or equal to 2. Write a program that prompts the user to enter a temperature and a wind speed. The program displays the wind-chill temperature if the input is valid, otherwise displays a message indicating whether the temperature and/or wind speed is invalid.

Comprehensives

3.21** (*Science: day of the week*) Zeller's congruence is an algorithm developed by Christian Zeller to calculate the day of the week. The formula is

$$h = \left(q + \left\lfloor \frac{26(m+1)}{10} \right\rfloor + k + \left\lfloor \frac{k}{4} \right\rfloor + \left\lfloor \frac{j}{4} \right\rfloor + 5j\right) \% 7$$

where

- `h` is the day of the week (0: Saturday, 1: Sunday, 2: Monday, 3: Tuesday, 4: Wednesday, 5: Thursday, 6: Friday).
- `q` is the day of the month.
- `m` is the month (3: March, 4: April, . . . , 12: December). January and February are counted as months 13 and 14 of the previous year.
- `j` is the century (i.e., $\left\lfloor \frac{year}{100} \right\rfloor$).
- `k` is the year of the century (i.e., $year \% 7$).

Write a program that prompts the user to enter a year, month, and day of the month, and displays the name of the day of the week. Here are some sample runs:

```
Enter year: (e.g., 2008): 2002 Enter
Enter month: 1-12: 3 Enter
Enter the day of the month: 1-31: 26 Enter
Day of the week is Tuesday
```



```
Enter year: (e.g., 2008): 2011 Enter
Enter month: 1-12: 5 Enter
Enter the day of the month: 1-31: 2 Enter
Day of the week is Thursday
```



(*Hint*: $\lfloor n \rfloor = (\text{int})n$ for a positive n. January and February are counted as 13 and 14 in the formula. So you need to convert the user input 1 to 13 and 2 to 14 for the month and change the year to the previous year.)

3.22** (*Geometry: point in a circle?*) Write a program that prompts the user to enter a point (`x`, `y`) and checks whether the point is within the circle centered at (`0`, `0`) with radius `10`. For example, (`4`, `5`) is inside the circle and (`9`, `9`) is outside the circle, as shown in Figure 3.7(a).

(*Hint*: A point is in the circle if its distance to (`0`, `0`) is less than or equal to `10`. The formula for computing the distance is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.) Two sample runs are shown below.)

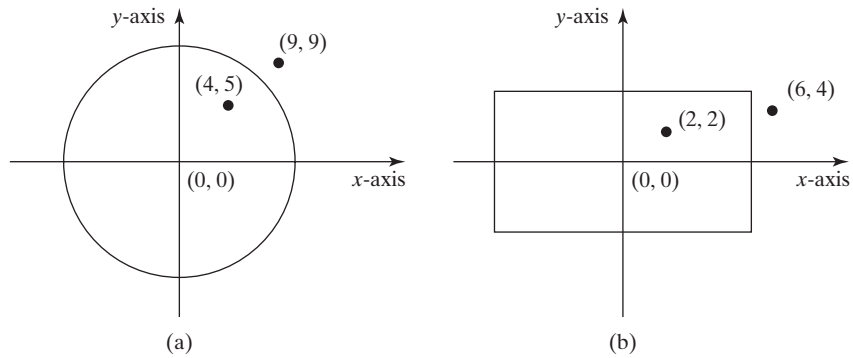


FIGURE 3.7 (a) Points inside and outside of the circle; (b) Points inside and outside of the rectangle.



```
Enter a point with two coordinates: 4 5 Enter
Point (4.0, 5.0) is in the circle
```



```
Enter a point with two coordinates: 9 9 Enter
Point (9.0, 9.0) is not in the circle
```

3.23** (*Geometry: point in a rectangle?*) Write a program that prompts the user to enter a point `(x, y)` and checks whether the point is within the rectangle centered at (`0`, `0`) with width `10` and height `5`. For example, (`2`, `2`) is inside the rectangle and (`6`, `4`) is outside the circle, as shown in Figure 3.7(b).

(*Hint*: A point is in the rectangle if its horizontal distance to (`0`, `0`) is less than or equal to `10 / 2` and its vertical distance to (`0`, `0`) is less than or equal to `5 / 2`.] Here are two sample runs. Two sample runs are shown below.)



```
Enter a point with two coordinates: 2 2 Enter
Point (2.0, 2.0) is in the rectangle
```



```
Enter a point with two coordinates: 6 4 Enter
Point (6.0, 4.0) is not in the rectangle
```

3.24** (*Game: picking a card*) Write a program that simulates picking a card from a deck of `52` cards. Your program should display the rank (`Ace`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `10`, `Jack`, `Queen`, `King`) and suit (`Clubs`, `Diamonds`, `Hearts`, `Spades`) of the card. Here is a sample run of the program:

```
The card you picked is Jack of Hearts
```



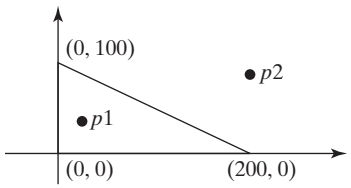
3.25** (*Computing the perimeter of a triangle*) Write a program that reads three edges for a triangle and computes the perimeter if the input is valid. Otherwise, display that the input is invalid. The input is valid if the sum of any two edges is greater than the third edge.

3.26 (*Using the* `&&`, `||` *and* `^` *operators*) Write a program that prompts the user to enter an integer and determines whether it is divisible by 5 and 6, whether it is divisible by 5 or 6, and whether it is divisible by 5 or 6, but not both. Here is a sample run of this program:

```
Enter an integer: 10 Enter
Is 10 divisible by 5 and 6? false
Is 10 divisible by 5 or 6? true
Is 10 divisible by 5 or 6, but not both? true
```



3.27** (*Geometry: points in triangle?*) Suppose a right triangle is placed in a plane as shown below. The right-angle point is placed at (0, 0), and the other two points are placed at (200, 0), and (0, 100). Write a program that prompts the user to enter a point with x- and y-coordinates and determines whether the point is inside the triangle. Here are the sample runs:




```
Enter a point's x- and y-coordinates: 100.5 25.5 Enter
The point is in the triangle
```



```
Enter a point's x- and y-coordinates: 100.5 50.5 Enter
The point is not in the triangle
```



3.28** (*Geometry: two rectangles*) Write a program that prompts the user to enter the center *x*-, *y*-coordinates, width, and height of two rectangles and determines whether the second rectangle is inside the first or overlaps with the first, as shown in Figure 3.8.

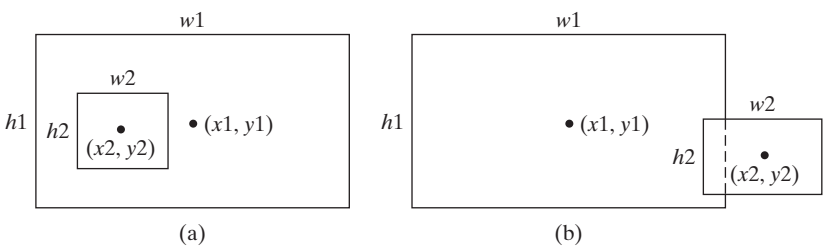


FIGURE 3.8 (a) A rectangle is inside another one. (b) A rectangle overlaps another one.

Here are the sample runs:



```
Enter r1's center x-, y-coordinates, width, and height:
  2.5 4 2.5 43 ↵Enter
Enter r2's center x-, y-coordinates, width, and height:
  1.5 5 0.5 3 ↵Enter
r2 is inside r1
```



```
Enter r1's center x-, y-coordinates, width, and height:
  1 2 3 5.5 ↵Enter
Enter r2's center x-, y-coordinates, width, and height:
  3 4 4.5 5 ↵Enter
r2 overlaps r1
```



```
Enter r1's center x-, y-coordinates, width, and height:
  1 2 3 3 ↵Enter
Enter r2's center x-, y-coordinates, width, and height:
  40 45 3 2 ↵Enter
r2 does not overlap r1
```

3.29** (*Geometry: two circles*) Write a program that prompts the user to enter the center coordinates and radii of two circles and determines whether the second circle is inside the first or overlaps with the first, as shown in Figure 3.9.

(*Hint*: circle2 is inside circle1 if the distance between the two centers <= `|r1 - r2|` and circle2 overlaps circle1 if the distance between the two centers <= `r1 + r2`.)

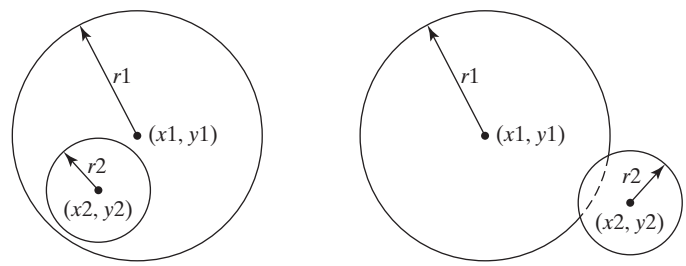


FIGURE 3.9 (a) A circle is inside another circle. (b) A circle overlaps another circle.

Here are the sample runs:



```
Enter circle1's center x-, y-coordinates, and radius:
  0.5 5.1 13 ↵Enter
Enter circle2's center x-, y-coordinates, and radius:
  1 1.7 4.5 ↵Enter
circle2 is inside circle1
```

```
Enter circle1's center x-, y-coordinates, and radius:
  3.4 5.7 5.5 ↵ Enter
Enter circle2's center x-, y-coordinates, and radius:
  6.7 3.5 3 ↵ Enter
circle2 overlaps circle1
```



```
Enter circle1's center x-, y-coordinates, and radius:
  3.4 5.5 1 ↵ Enter
Enter circle2's center x-, y-coordinates, and radius:
  5.5 7.2 1 ↵ Enter
circle2 does not overlap circle1
```



This page intentionally left blank

CHAPTER 4

LOOPS

Objectives

- To write programs for executing statements repeatedly using a `while` loop (§4.2).
- To develop a program for `GuessNumber` (§4.2.1).
- To follow the loop design strategy to develop loops (§4.2.2).
- To develop a program for `SubtractionQuizLoop` (§4.2.3).
- To control a loop with a sentinel value (§4.2.4).
- To obtain large input from a file using input redirection rather than typing from the keyboard (§4.2.4).
- To write loops using `do-while` statements (§4.3).
- To write loops using `for` statements (§4.4).
- To discover the similarities and differences of three types of loop statements (§4.5).
- To write nested loops (§4.6).
- To learn the techniques for minimizing numerical errors (§4.7).
- To learn loops from a variety of examples (`GCD`, `FutureTuition`, `MonteCarloSimulation`) (§4.8).
- To implement program control with `break` and `continue` (§4.9).
- (GUI) To control a loop with a confirmation dialog (§4.10).



4.1 Introduction

Suppose that you need to print a string (e.g., `"Welcome to Java!"`) a hundred times. It would be tedious to have to write the following statement a hundred times:

100 times

```
System.out.println("Welcome to Java!");
System.out.println("Welcome to Java!");
...
System.out.println("Welcome to Java!");
```

problem

why loop?

So, how do you solve this problem?

Java provides a powerful construct called a *loop* that controls how many times an operation or a sequence of operations is performed in succession. Using a loop statement, you simply tell the computer to print a string a hundred times without having to code the print statement a hundred times, as follows:

```
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java!");
  count++;
}
```

The variable `count` is initially `0`. The loop checks whether (`count < 100`) is `true`. If so, it executes the loop body to print the message `"Welcome to Java!"` and increments `count` by `1`. It repeatedly executes the loop body until (`count < 100`) becomes `false`. When (`count < 100`) is `false` (i.e., when `count` reaches `100`), the loop terminates and the next statement after the loop statement is executed.

Loops are constructs that control repeated executions of a block of statements. The concept of looping is fundamental to programming. Java provides three types of loop statements: `while` loops, `do-while` loops, and `for` loops.

4.2 The `while` Loop

The syntax for the `while` loop is as follows:

while loop

```
while (loop-continuation-condition) {
  // Loop body
  Statement(s);
}
```

Figure 4.1(a) shows the `while`-loop flow chart. The part of the loop that contains the statements to be repeated is called the *loop body*. A one-time execution of a loop body is referred to as an *iteration of the loop*. Each loop contains a loop-continuation-condition, a Boolean expression that controls the execution of the body. It is evaluated each time to determine if the loop body is executed. If its evaluation is `true`, the loop body is executed; if its evaluation is `false`, the entire loop terminates and the program control turns to the statement that follows the `while` loop.

loop body

iteration

The loop for printing `Welcome to Java!` a hundred times introduced in the preceding section is an example of a `while` loop. Its flow chart is shown in Figure 4.1(b). The `loop-continuation-condition` is `(count < 100)` and loop body contains two statements as shown below:

loop-continuation-condition

loop body

```
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java!");
  count++;
}
```

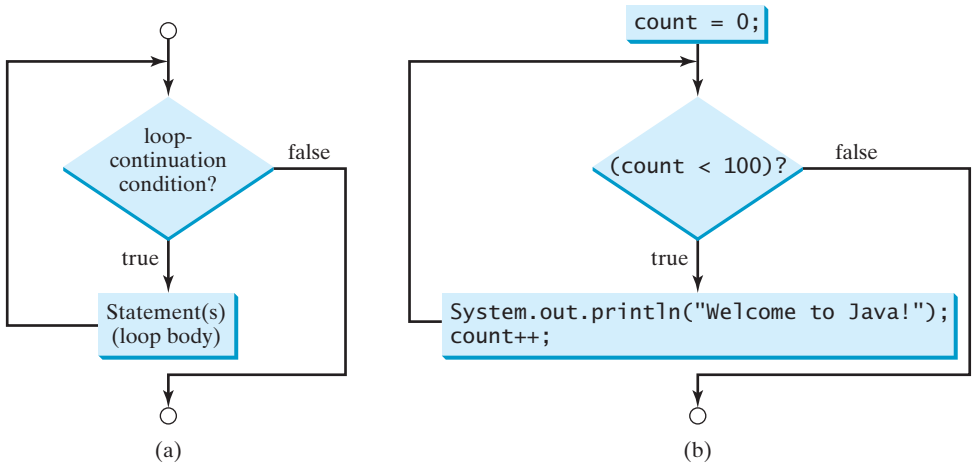


FIGURE 4.1 The `while` loop repeatedly executes the statements in the loop body when the `loop-continuation-condition` evaluates to `true`.

In this example, you know exactly how many times the loop body needs to be executed. So a control variable `count` is used to count the number of executions. This type of loop is known as a *counter-controlled loop*.

counter-controlled loop



Note

The `loop-continuation-condition` must always appear inside the parentheses. The braces enclosing the loop body can be omitted only if the loop body contains one or no statement.

Here is another example to help understand how a loop works.

```
int sum = 0, i = 1;
while (i < 10) {
  sum = sum + i;
  i++;
}
System.out.println("sum is " + sum); // sum is 45
```

If `i < 10` is `true`, the program adds `i` to `sum`. Variable `i` is initially set to `1`, then incremented to `2`, `3`, and up to `10`. When `i` is `10`, `i < 10` is `false`, the loop exits. So, the sum is `1 + 2 + 3 + ... + 9 = 45`.

What happens if the loop is mistakenly written as follows:

```
int sum = 0, i = 1;
while (i < 10) {
  sum = sum + i;
}
```

This loop is infinite, because `i` is always `1` and `i < 10` will always be `true`.



Caution

Make sure that the `loop-continuation-condition` eventually becomes `false` so that the program will terminate. A common programming error involves infinite loops. That is, the program cannot terminate because of a mistake in the `loop-continuation-condition`.

infinite loop

Programmers often make mistakes to execute a loop one more or less time. This is commonly known as the *off-by-one error*. For example, the following loop displays `Welcome to Java` 101 times rather than 100 times. The error lies in the condition, which should be `count < 100` rather than `count <= 100`.

off-by-one error

```
int count = 0;
while (count <= 100) {
  System.out.println("Welcome to Java!");
  count++;
}
```

4.2.1 Problem: Guessing Numbers



Video Note
Guess a number

The problem is to guess what a number a computer has in mind. You will write a program that randomly generates an integer between **0** and **100**, inclusive. The program prompts the user to enter a number continuously until the number matches the randomly generated number. For each user input, the program tells the user whether the input is too low or too high, so the user can make the next guess intelligently. Here is a sample run:



```
Guess a magic number between 0 and 100

Enter your guess: 50 Enter
Your guess is too high

Enter your guess: 25 Enter
Your guess is too high

Enter your guess: 12 Enter
Your guess is too high

Enter your guess: 6 Enter
Your guess is too low

Enter your guess: 9 Enter
Yes, the number is 9
```

intelligent guess

The magic number is between **0** and **100**. To minimize the number of guesses, enter **50** first. If your guess is too high, the magic number is between **0** and **49**. If your guess is too low, the magic number is between **51** and **100**. So, you can eliminate half of the numbers from further consideration after one guess.

think before coding

How do you write this program? Do you immediately begin coding? No. It is important to *think before coding*. Think how you would solve the problem without writing a program. You need first to generate a random number between **0** and **100**, inclusive, then to prompt the user to enter a guess, and then to compare the guess with the random number.

code incrementally

It is a good practice to *code incrementally* one step at a time. For programs involving loops, if you don't know how to write a loop right away, you may first write the code for executing the loop one time, and then figure out how to repeatedly execute the code in a loop. For this program, you may create an initial draft, as shown in Listing 4.1:

LISTING 4.1 GuessNumberOneTime.java

```
1  import java.util.Scanner;
2
3  public class GuessNumberOneTime {
4    public static void main(String[] args) {
5      // Generate a random number to be guessed
6      int number = (int)(Math.random() * 101);
7
8      Scanner input = new Scanner(System.in);
9      System.out.println("Guess a magic number between 0 and 100");
10
11     // Prompt the user to guess the number
12     System.out.print("\nEnter your guess: ");
13     int guess = input.nextInt();
```

generate a number

enter a guess

```
14
15      if (guess == number)
16        System.out.println("Yes, the number is " + number);          correct guess?
17      else if (guess > number)
18        System.out.println("Your guess is too high");                too high?
19      else
20        System.out.println("Your guess is too low");                 too low?
21    }
22  }
```

When you run this program, it prompts the user to enter a guess only once. To let the user enter a guess repeatedly, you may put the code in lines 11–20 in a loop as follows:

```
while (true) {
  // Prompt the user to guess the number
  System.out.print("\nEnter your guess: ");
  guess = input.nextInt();

  if (guess == number)
    System.out.println("Yes, the number is " + number);
  else if (guess > number)
    System.out.println("Your guess is too high");
  else
    System.out.println("Your guess is too low");
} // End of loop
```

This loop repeatedly prompts the user to enter a guess. However, this loop is not correct, because it never terminates. When `guess` matches `number`, the loop should end. So, the loop can be revised as follows:

```
while (guess != number) {
  // Prompt the user to guess the number
  System.out.print("\nEnter your guess: ");
  guess = input.nextInt();

  if (guess == number)
    System.out.println("Yes, the number is " + number);
  else if (guess > number)
    System.out.println("Your guess is too high");
  else
    System.out.println("Your guess is too low");
} // End of loop
```

The complete code is given in Listing 4.2.

LISTING 4.2 GuessNumber.java

```
1  import java.util.Scanner;
2
3  public class GuessNumber {
4    public static void main(String[] args) {
5      // Generate a random number to be guessed
6      int number = (int)(Math.random() * 101);                        generate a number
7
8      Scanner input = new Scanner(System.in);
9      System.out.println("Guess a magic number between 0 and 100");
10
11     int guess = -1;
12     while (guess != number) {
```

enter a guess

too high?

too low?

```
13        // Prompt the user to guess the number
14        System.out.print("\nEnter your guess: ");
15        guess = input.nextInt();
16
17        if (guess == number)
18          System.out.println("Yes, the number is " + number);
19        else if (guess > number)
20          System.out.println("Your guess is too high");
21        else
22          System.out.println("Your guess is too low");
23      } // End of loop
24    }
25 }
```



	line#	number	guess	output
	6	8		
	11		-1	
iteration 1	15		50	
	20			Your guess is too high
iteration 2	15		25	
	20			Your guess is too high
iteration 3	15		12	
	20			Your guess is too high
iteration 4	15		6	
	22			Your guess is too low
iteration 5	15		9	
	20			Yes, the number is 9

The program generates the magic number in line 6 and prompts the user to enter a guess continuously in a loop (lines 12–23). For each guess, the program checks whether the guess is correct, too high, or too low (lines 17–22). When the guess is correct, the program exits the loop (line 12). Note that `guess` is initialized to `-1`. Initializing it to a value between `0` and `100` would be wrong, because that could be the number to be guessed.

4.2.2 Loop Design Strategies

Writing a correct loop is not an easy task for novice programmers. Consider three steps when writing a loop.

Step 1: Identify the statements that need to be repeated.

Step 2: Wrap these statements in a loop like this:

```
while (true) {
  Statements;
}
```

Step 3: Code the loop-continuation-condition and add appropriate statements for controlling the loop.

```
while (loop-continuation-condition) {
  Statements;
  Additional statements for controlling the loop;
}
```

4.2.3 Problem: An Advanced Math Learning Tool



Video Note
Multiple subtraction quiz

The Math subtraction learning tool program in Listing 3.4, SubtractionQuiz.java, generates just one question for each run. You can use a loop to generate questions repeatedly. How do you write the code to generate five questions? Follow the loop design strategy. First identify the statements that need to be repeated. These are the statements for obtaining two random numbers, prompting the user with a subtraction question, and grading the question. Second, wrap the statements in a loop. Third, add a loop control variable and the loop-continuation-condition to execute the loop five times.

Listing 4.3 gives a program that generates five questions and, after a student answers all five, reports the number of correct answers. The program also displays the time spent on the test and lists all the questions.

LISTING 4.3 SubtractionQuizLoop.java

```
1  import java.util.Scanner;
2
3  public class SubtractionQuizLoop {
4    public static void main(String[] args) {
5      final int NUMBER_OF_QUESTIONS = 5; // Number of questions
6      int correctCount = 0; // Count the number of correct answers
7      int count = 0; // Count the number of questions
8      long startTime = System.currentTimeMillis();                    get start time
9      String output = ""; // output string is initially empty
10     Scanner input = new Scanner(System.in);
11
12     while (count < NUMBER_OF_QUESTIONS) {                           loop
13       // 1. Generate two random single-digit integers
14       int number1 = (int)(Math.random() * 10);
15       int number2 = (int)(Math.random() * 10);
16
17       // 2. If number1 < number2, swap number1 with number2
18       if (number1 < number2) {
19         int temp = number1;
20         number1 = number2;
21         number2 = temp;
22       }
23
24       // 3. Prompt the student to answer "What is number1 - number2?"
25       System.out.print(                                             display a question
26         "What is " + number1 + " - " + number2 + "? ");
27       int answer = input.nextInt();
28
29       // 4. Grade the answer and display the result
30       if (number1 - number2 == answer) {                            grade an answer
31         System.out.println("You are correct!");
32         correctCount++;                                             increase correct count
33       }
34       else
35         System.out.println("Your answer is wrong.\n" + number1
36           + " - " + number2 + " should be " + (number1 - number2));
37
```

increase control variable

prepare output

end loop

get end time

test time

display result

```
38        // Increase the count
39        count++;
40
41        output += "\n" + number1 + "-" + number2 + "=" + answer +
42          ((number1 - number2 == answer) ? " correct" : " wrong");
43      }
44
45      long endTime = System.currentTimeMillis();
46      long testTime = endTime - startTime;
47
48      System.out.println("Correct count is " + correctCount +
49        "\nTest time is " + testTime / 1000 + " seconds\n" + output);
50    }
51 }
```



```
What is 9 - 2? 7 Enter
You are correct!

What is 3 - 0? 3 Enter
You are correct!

What is 3 - 2? 1 Enter
You are correct!

What is 7 - 4? 4 Enter
Your answer is wrong.
7 - 4 should be 3

What is 7 - 5? 4 Enter
Your answer is wrong.
7 - 5 should be 2

Correct count is 3
Test time is 1021 seconds

9-2=7 correct
3-0=3 correct
3-2=1 correct
7-4=4 wrong
7-5=4 wrong
```

The program uses the control variable **count** to control the execution of the loop. **count** is initially **0** (line 7) and is increased by **1** in each iteration (line 39). A subtraction question is displayed and processed in each iteration. The program obtains the time before the test starts in line 8 and the time after the test ends in line 45, and computes the test time in line 46. The test time is in milliseconds and is converted to seconds in line 49.

4.2.4 Controlling a Loop with a Sentinel Value

sentinel value

Another common technique for controlling a loop is to designate a special value when reading and processing a set of values. This special input value, known as a *sentinel value*, signifies the end of the loop. A loop that uses a sentinel value to control its execution is called a *sentinel-controlled loop*.

Listing 4.4 writes a program that reads and calculates the sum of an unspecified number of integers. The input **0** signifies the end of the input. Do you need to declare a new variable for each input value? No. Just use one variable named **data** (line 12) to store the input value and use a variable named **sum** (line 15) to store the total. Whenever a value is read, assign it to **data** and, if it is not zero, add it to **sum** (line 17).

LISTING 4.4 SentinelValue.java

```
1 import java.util.Scanner;
2
3 public class SentinelValue {
4   /** Main method */
5   public static void main(String[] args) {
6     // Create a Scanner
7     Scanner input = new Scanner(System.in);
8
9     // Read an initial data
10    System.out.print(
11      "Enter an int value (the program exits if the input is 0): ");
12    int data = input.nextInt();                                    input
13
14    // Keep reading data until the input is 0
15    int sum = 0;
16    while (data != 0) {                                            loop
17      sum += data;
18
19      // Read the next data
20      System.out.print(
21        "Enter an int value (the program exits if the input is 0): ");
22      data = input.nextInt();
23    }                                                              end of loop
24
25    System.out.println("The sum is " + sum);                      display result
26  }
27 }
```

```
Enter an int value (the program exits if the input is 0): 2 Enter
Enter an int value (the program exits if the input is 0): 3 Enter
Enter an int value (the program exits if the input is 0): 4 Enter
Enter an int value (the program exits if the input is 0): 0 Enter
The sum is 9
```





	line#	data	sum	output
	12	2		
	15		0	
iteration 1	17		2	
	22	3		
iteration 2	17		5	
	22	4		
iteration 3	17		9	
	22	0		
	25			The sum is 9

If **data** is not **0**, it is added to **sum** (line 17) and the next item of input data is read (lines 20–22). If **data** is **0**, the loop body is no longer executed and the **while** loop terminates. The input value **0** is the sentinel value for this loop. Note that if the first input read is **0**, the loop body never executes, and the resulting sum is **0**.



Caution

numeric error

Don't use floating-point values for equality checking in a loop control. Since floating-point values are approximations for some values, using them could result in imprecise counter values and inaccurate results.

Consider the following code for computing `1 + 0.9 + 0.8 + ... + 0.1`:

```
double item = 1; double sum = 0;
while (item != 0) { // No guarantee item will be 0
  sum += item;
  item -= 0.1;
}
System.out.println(sum);
```

Variable `item` starts with `1` and is reduced by `0.1` every time the loop body is executed. The loop should terminate when `item` becomes `0`. However, there is no guarantee that item will be exactly `0`, because the floating-point arithmetic is approximated. This loop seems OK on the surface, but it is actually an infinite loop.

4.2.5 Input and Output Redirections

In the preceding example, if you have a large number of data to enter, it would be cumbersome to type from the keyboard. You may store the data separated by whitespaces in a text file, say input.txt, and run the program using the following command:

```
java SentinelValue < input.txt
```

input redirection

This command is called *input redirection*. The program takes the input from the file input.txt rather than having the user to type the data from the keyboard at runtime. Suppose the contents of the file are

```
2 3 4 5 6 7 8 9 12 23 32
23 45 67 89 92 12 34 35 3 1 2 4 0
```

The program should get `sum` to be `518`.

output redirection

Similarly, there is *output redirection*, which sends the output to a file rather than displaying it on the console. The command for output redirection is:

```
java ClassName > output.txt
```

Input and output redirection can be used in the same command. For example, the following command gets input from input.txt and sends output to output.txt:

```
java SentinelValue < input.txt > output.txt
```

Please run the program and see what contents are in output.txt.

4.3 The `do-while` Loop

The `do-while` loop is a variation of the `while` loop. Its syntax is given below:

do-while loop

```
do {
  // Loop body;
  Statement(s);
} while (loop-continuation-condition);
```

Its execution flow chart is shown in Figure 4.2.

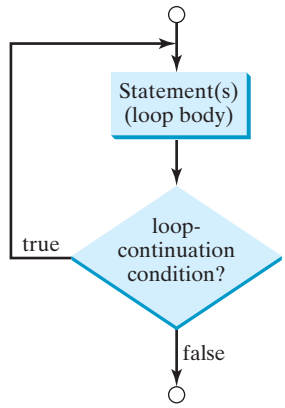


FIGURE 4.2 The `do-while` loop executes the loop body first, then checks the `loop-continuation-condition` to determine whether to continue or terminate the loop.

The loop body is executed first. Then the `loop-continuation-condition` is evaluated. If the evaluation is `true`, the loop body is executed again; if it is `false`, the `do-while` loop terminates. The difference between a `while` loop and a `do-while` loop is the order in which the `loop-continuation-condition` is evaluated and the loop body executed. The `while` loop and the `do-while` loop have equal expressive power. Sometimes one is a more convenient choice than the other. For example, you can rewrite the `while` loop in Listing 4.4 using a `do-while` loop, as shown in Listing 4.5:

LISTING 4.5 TestDoWhile.java

```
1  import java.util.Scanner;
2
3  public class TestDoWhile {
4    /** Main method */
5    public static void main(String[] args) {
6      int data;
7      int sum = 0;
8
9      // Create a Scanner
10     Scanner input = new Scanner(System.in);
11
12     // Keep reading data until the input is 0
13     do {                                                          loop
14       // Read the next data
15       System.out.print(
16         "Enter an int value (the program exits if the input is 0): ");
17       data = input.nextInt();
18
19       sum += data;
20     } while (data != 0);                                          end loop
21
22     System.out.println("The sum is " + sum);
23   }
24 }
```

```
Enter an int value (the program exits if the input is 0): 3 Enter
Enter an int value (the program exits if the input is 0): 5 Enter
Enter an int value (the program exits if the input is 0): 6 Enter
Enter an int value (the program exits if the input is 0): 0 Enter
The sum is 14
```





Tip

Use the `do-while` loop if you have statements inside the loop that must be executed *at least once*, as in the case of the `do-while` loop in the preceding `TestDoWhile` program. These statements must appear before the loop as well as inside it if you use a `while` loop.

4.4 The for Loop

Often you write a loop in the following common form:

```
i = initialValue;  // Initialize loop control variable
while (i < endValue) {
  // Loop body
  ...
  i++; // Adjust loop control variable
}
```

A `for` loop can be used to simplify the proceding loop:

```
for (i = initialValue; i < endValue; i++) {
  // Loop body
  ...
}
```

In general, the syntax of a `for` loop is as shown below:

for loop

```
for (initial-action; loop-continuation-condition;
     action-after-each-iteration) {
  // Loop body;
  Statement(s);
}
```

The flow chart of the `for` loop is shown in Figure 4.3(a).

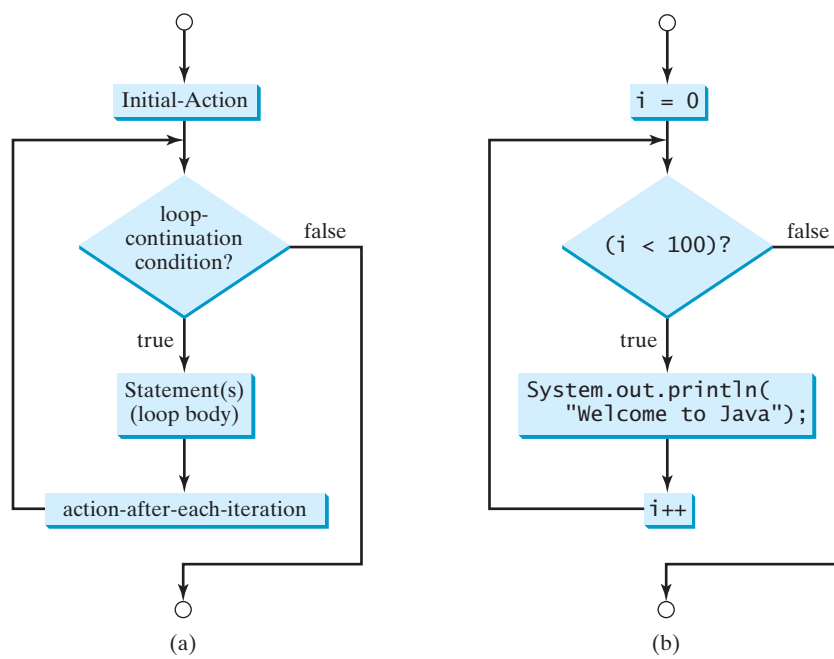


FIGURE 4.3 A `for` loop performs an initial action once, then repeatedly executes the statements in the loop body, and performs an action after an iteration when the `loop-continuation-condition` evaluates to `true`.

The **for** loop statement starts with the keyword **for**, followed by a pair of parentheses enclosing the control structure of the loop. This structure consists of **initial-action**, **loop-continuation-condition**, and **action-after-each-iteration**. The control structure is followed by the loop body enclosed inside braces. The **initial-action**, **loop-continuation-condition**, and **action-after-each-iteration** are separated by semicolons.

A **for** loop generally uses a variable to control how many times the loop body is executed and when the loop terminates. This variable is referred to as a *control variable*. The **initial-action** often initializes a control variable, the **action-after-each-iteration** usually increments or decrements the control variable, and the **loop-continuation-condition** tests whether the control variable has reached a termination value. For example, the following **for** loop prints **Welcome to Java!** a hundred times:

control variable

```
int i;
for (i = 0; i < 100; i++) {
  System.out.println("Welcome to Java!");
}
```

The flow chart of the statement is shown in Figure 4.3(b). The **for** loop initializes **i** to **0**, then repeatedly executes the **println** statement and evaluates **i++** while **i** is less than **100**.

The **initial-action**, **i = 0**, initializes the control variable, **i**. The **loop-continuation-condition**, **i < 100**, is a Boolean expression. The expression is evaluated right after the initialization and at the beginning of each iteration. If this condition is **true**, the loop body is executed. If it is **false**, the loop terminates and the program control turns to the line following the loop.

initial-action

The **action-after-each-iteration**, **i++**, is a statement that adjusts the control variable. This statement is executed after each iteration. It increments the control variable. Eventually, the value of the control variable should force the **loop-continuation-condition** to become **false**. Otherwise the loop is infinite.

action-after-each-iteration

The loop control variable can be declared and initialized in the for loop. Here is an example:

```
for (int i = 0; i < 100; i++) {
  System.out.println("Welcome to Java!");
}
```

If there is only one statement in the loop body, as in this example, the braces can be omitted.

omitting braces



Tip

The control variable must be declared inside the control structure of the loop or before the loop. If the loop control variable is used only in the loop, and not elsewhere, it is good programming practice to declare it in the **initial-action** of the **for** loop. If the variable is declared inside the loop control structure, it cannot be referenced outside the loop. In the preceding code, for example, you cannot reference **i** outside the **for** loop, because it is declared inside the **for** loop.

declare control variable



Note

The **initial-action** in a **for** loop can be a list of zero or more comma-separated variable declaration statements or assignment expressions. For example,

for loop variations

```
for (int i = 0, j = 0; (i + j < 10); i++, j++) {
  // Do something
}
```

The **action-after-each-iteration** in a **for** loop can be a list of zero or more comma-separated statements. For example,

```
for (int i = 1; i < 100; System.out.println(i), i++);
```

This example is correct, but it is a bad example, because it makes the code difficult to read. Normally, you declare and initialize a control variable as an initial action and increment or decrement the control variable as an action after each iteration.



Note

If the `loop-continuation-condition` in a `for` loop is omitted, it is implicitly `true`. Thus the statement given below in (a), which is an infinite loop, is the same as in (b). To avoid confusion, though, it is better to use the equivalent loop in (c):

```
for ( ; ; ) {
  // Do something
}
```

(a)

Equivalent

```
for ( ; true; ) {
  // Do something
}
```

(b)

Equivalent

This is better

```
while (true) {
  // Do something
}
```

(c)

4.5 Which Loop to Use?

pretest loop
posttest loop

The `while` loop and `for` loop are called *pretest loops* because the continuation condition is checked before the loop body is executed. The `do-while` loop is called a *posttest loop* because the condition is checked after the loop body is executed. The three forms of loop statements, `while`, `do-while`, and `for`, are expressively equivalent; that is, you can write a loop in any of these three forms. For example, a `while` loop in (a) in the following figure can always be converted into the `for` loop in (b):

```
while (loop-continuation-condition) {
  // Loop body
}
```

(a)

Equivalent

```
for ( ; loop-continuation-condition; ) {
  // Loop body
}
```

(b)

A `for` loop in (a) in the next figure can generally be converted into the `while` loop in (b) except in certain special cases (see Review Question 4.17 for such a case):

```
for (initial-action;
     loop-continuation-condition;
     action-after-each-iteration) {
  // Loop body;
}
```

(a)

Equivalent

```
initial-action;
while (loop-continuation-condition) {
  // Loop body;
  action-after-each-iteration;
}
```

(b)

Use the loop statement that is most intuitive and comfortable for you. In general, a `for` loop may be used if the number of repetitions is known in advance, as, for example, when you need to print a message a hundred times. A `while` loop may be used if the number of repetitions is not fixed, as in the case of reading the numbers until the input is `0`. A `do-while` loop can be used to replace a `while` loop if the loop body has to be executed before the continuation condition is tested.



Caution

Adding a semicolon at the end of the `for` clause before the loop body is a common mistake, as shown below in (a). In (a), the semicolon signifies the end of the loop prematurely. The loop body is actually empty, as shown in (b). (a) and (b) are equivalent.

Error

```
for (int i = 0; i < 10; i++);
{
  System.out.println("i is " + i);
}
```

(a)

Empty Body

```
for (int i = 0; i < 10; i++) { };
{
  System.out.println("i is " + i);
}
```

(b)

Similarly, the loop in (c) is also wrong. (c) is equivalent to (d).

Error

```
int i = 0;
while (i < 10);
{
  System.out.println("i is " + i);
  i++;
}
```

(c)

Empty Body

```
int i = 0;
while (i < 10) { };
{
  System.out.println("i is " + i);
  i++;
}
```

(d)

These errors often occur when you use the next-line block style. Using the end-of-line block style can avoid errors of this type.

In the case of the **do-while** loop, the semicolon is needed to end the loop.

```
int i = 0;
do {
  System.out.println("i is " + i);
  i++;
} while (i < 10);
```

Correct

4.6 Nested Loops

Nested loops consist of an outer loop and one or more inner loops. Each time the outer loop is repeated, the inner loops are reentered, and started anew.

Listing 4.6 presents a program that uses nested **for** loops to print a multiplication table.

LISTING 4.6 MultiplicationTable.java

```
1  public class MultiplicationTable {
2    /** Main method */
3    public static void main(String[] args) {
4      // Display the table heading
5      System.out.println("           Multiplication Table");                table title
6
7      // Display the number title
8      System.out.print("    ");
9      for (int j = 1; j <= 9; j++)
10       System.out.print("   " + j);
11
12     System.out.println("\n---------------------------------------");
13
14     // Print table body
15     for (int i = 1; i <= 9; i++) {                                          outer loop
16       System.out.print(i + " | ");
17       for (int j = 1; j <= 9; j++) {                                        inner loop
18         // Display the product and align properly
19         System.out.printf("%4d", i * j);
20       }
21       System.out.println()
22     }
23   }
24 }
```



```
            Multiplication Table
       1   2   3   4   5   6   7   8   9
   ----------------------------------------
1  |   1   2   3   4   5   6   7   8   9
2  |   2   4   6   8  10  12  14  16  18
3  |   3   6   9  12  15  18  21  24  27
4  |   4   8  12  16  20  24  28  32  36
5  |   5  10  15  20  25  30  35  40  45
6  |   6  12  18  24  30  36  42  48  54
7  |   7  14  21  28  35  42  49  56  63
8  |   8  16  24  32  40  48  56  64  72
9  |   9  18  27  36  45  54  63  72  81
```

The program displays a title (line 5) on the first line in the output. The first `for` loop (lines 9–10) displays the numbers `1` through `9` on the second line. A dash (`-`) line is displayed on the third line (line 12).

The next loop (lines 15–22) is a nested `for` loop with the control variable `i` in the outer loop and `j` in the inner loop. For each `i`, the product `i * j` is displayed on a line in the inner loop, with `j` being `1`, `2`, `3`, ..., `9`.

4.7 Minimizing Numeric Errors



Video Note
Minimize numeric errors

Numeric errors involving floating-point numbers are inevitable. This section discusses how to minimize such errors through an example.

Listing 4.7 presents an example summing a series that starts with `0.01` and ends with `1.0`. The numbers in the series will increment by `0.01`, as follows: `0.01 + 0.02 + 0.03` and so on.

LISTING 4.7 TestSum.java

```
1 public class TestSum {
2   public static void main(String[] args) {
3     // Initialize sum
4     float sum = 0;
5
6     // Add 0.01, 0.02, ..., 0.99, 1 to sum
7     for (float i = 0.01f; i <= 1.0f; i = i + 0.01f)
8       sum += i;
9
10    // Display result
11    System.out.println("The sum is " + sum);
12  }
13 }
```

loop



```
The sum is 50.499985
```

The `for` loop (lines 7–8) repeatedly adds the control variable `i` to `sum`. This variable, which begins with `0.01`, is incremented by `0.01` after each iteration. The loop terminates when `i` exceeds `1.0`.

The `for` loop initial action can be any statement, but it is often used to initialize a control variable. From this example, you can see that a control variable can be a `float` type. In fact, it can be any data type.

The exact `sum` should be `50.50`, but the answer is `50.499985`. The result is imprecise because computers use a fixed number of bits to represent floating-point numbers, and thus they cannot represent some floating-point numbers exactly. If you change `float` in the program to `double`, as follows, you should see a slight improvement in precision, because a `double` variable takes 64 bits, whereas a `float` variable takes 32.

double precision

```
// Initialize sum
double sum = 0;

// Add 0.01, 0.02, ..., 0.99, 1 to sum
for (double i = 0.01; i <= 1.0; i = i + 0.01)
  sum += i;
```

However, you will be stunned to see that the result is actually `49.50000000000003`. What went wrong? If you print out `i` for each iteration in the loop, you will see that the last `i` is slightly larger than `1` (not exactly `1`). This causes the last `i` not to be added into `sum`. The fundamental problem is that the floating-point numbers are represented by approximation. To fix the problem, use an integer count to ensure that all the numbers are added to `sum`. Here is the new loop:

numeric error

```
double currentValue = 0.01;

for (int count = 0; count < 100; count++) {
  sum += currentValue;
  currentValue += 0.01;
}
```

After this loop, `sum` is `50.50000000000003`. This loop adds the numbers from small to big. What happens if you add numbers from big to small (i.e., `1.0`, `0.99`, `0.98`, ..., `0.02`, `0.01` in this order) as follows:

```
double currentValue = 1.0;

for (int count = 0; count < 100; count++) {
  sum += currentValue;
  currentValue -= 0.01;
}
```

After this loop, `sum` is `50.49999999999995`. Adding from big to small is less accurate than adding from small to big. This phenomenon is an artifact of the finite-precision arithmetic. Adding a very small number to a very big number can have no effect if the result requires more precision than the variable can store. For example, the inaccurate result of `100000000.0 + 0.000000001` is `100000000.0`. To obtain more accurate results, carefully select the order of computation. Adding the smaller numbers before the big numbers is one way to minimize error.

avoiding numeric error

4.8 Case Studies

Loops are fundamental in programming. The ability to write loops is essential in learning Java programming. *If you can write programs using loops, you know how to program!* For this reason, this section presents three additional examples of solving problems using loops.

4.8.1 Problem: Finding the Greatest Common Divisor

The greatest common divisor of two integers `4` and `2` is `2`. The greatest common divisor of two integers `16` and `24` is `8`. How do you find the greatest common divisor? Let the two input integers be `n1` and `n2`. You know that number `1` is a common divisor, but it may not be the

gcd

greatest common divisor. So, you can check whether k (for k = 2, 3, 4, and so on) is a common divisor for n1 and n2, until k is greater than n1 or n2. Store the common divisor in a variable named gcd. Initially, gcd is 1. Whenever a new common divisor is found, it becomes the new gcd. When you have checked all the possible common divisors from 2 up to n1 or n2, the value in variable gcd is the greatest common divisor. The idea can be translated into the following loop:

```
int gcd = 1; // Initial gcd is 1
int k = 2; // Possible gcd

while (k <= n1 && k <= n2) {
  if (n1 % k == 0 && n2 % k == 0)
    gcd = k; // Update gcd
  k++; // Next possible gcd
}

// After the loop, gcd is the greatest common divisor for n1 and n2
```

Listing 4.8 presents the program that prompts the user to enter two positive integers and finds their greatest common divisor.

LISTING 4.8 GreatestCommonDivisor.java

```
1 import java.util.Scanner;
2
3 public class GreatestCommonDivisor {
4   /** Main method */
5   public static void main(String[] args) {
6     // Create a Scanner
7     Scanner input = new Scanner(System.in);
8
9     // Prompt the user to enter two integers
10    System.out.print("Enter first integer: ");
11    int n1 = input.nextInt();
12    System.out.print("Enter second integer: ");
13    int n2 = input.nextInt();
14
15    int gcd = 1; // Initial gcd is 1
16    int k = 2; // Possible gcd
17    while (k <= n1 && k <= n2) {
18      if (n1 % k == 0 && n2 % k == 0)
19        gcd = k; // Update gcd
20      k++;
21    }
22
23    System.out.println("The greatest common divisor for " + n1 +
24      " and " + n2 + " is " + gcd);
25  }
26 }
```

input

input

gcd

check divisor

output



```
Enter first integer: 125 Enter
Enter second integer: 2525 Enter
The greatest common divisor for 125 and 2525 is 25
```

How did you write this program? Did you immediately begin to write the code? No. It is important to *think before you type*. Thinking enables you to generate a logical solution for the problem without concern about how to write the code. Once you have a logical solution, type

think before you type

the code to translate the solution into a Java program. The translation is not unique. For example, you could use a `for` loop to rewrite the code as follows:

```
for (int k = 2; k <= n1 && k <= n2; k++) {
  if (n1 % k == 0 && n2 % k == 0)
    gcd = k;
}
```

A problem often has multiple solutions. The gcd problem can be solved in many ways. Exercise 4.15 suggests another solution. A more efficient solution is to use the classic Euclidean algorithm. See http://www.cut-the-knot.org/blue/Euclid.shtml for more information.

multiple solutions

You might think that a divisor for a number `n1` cannot be greater than `n1 / 2`. So you would attempt to improve the program using the following loop:

erroneous solutions

```
for (int k = 2; k <= n1 / 2 && k <= n2 / 2; k++) {
  if (n1 % k == 0 && n2 % k == 0)
    gcd = k;
}
```

This revision is wrong. Can you find the reason? See Review Question 4.14 for the answer.

4.8.2 Problem: Predicating the Future Tuition

Suppose that the tuition for a university is `$10,000` this year and tuition increases `7%` every year. In how many years will the tuition be doubled?

Before you can write a program to solve this problem, first consider how to solve it by hand. The tuition for the second year is the tuition for the first year * `1.07`. The tuition for a future year is the tuition of its preceding year * `1.07`. So, the tuition for each year can be computed as follows:

```
double tuition = 10000;   int year = 1  // Year 1
tuition = tuition * 1.07; year++;       // Year 2
tuition = tuition * 1.07; year++;       // Year 3
tuition = tuition * 1.07; year++;       // Year 4
...
```

Keep computing tuition for a new year until it is at least `20000`. By then you will know how many years it will take for the tuition to be doubled. You can now translate the logic into the following loop:

```
double tuition = 10000;   // Year 1
int year = 1;
while (tuition < 20000) {
  tuition = tuition * 1.07;
  year++;
}
```

The complete program is shown in Listing 4.9.

LISTING 4.9 FutureTuition.java

```
1 public class FutureTuition {
2   public static void main(String[] args) {
3     double tuition = 10000;   // Year 1
4     int year = 1;
5     while (tuition < 20000) {
```

loop

next year's tuition

```
6       tuition = tuition * 1.07;
7       year++;
8     }
9
10    System.out.println("Tuition will be doubled in "
11      + year + " years");
12  }
13 }
```



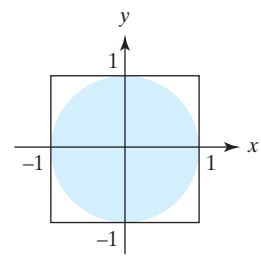
```
Tuition will be doubled in 12 years
```

The `while` loop (lines 5–8) is used to repeatedly compute the tuition for a new year. The loop terminates when tuition is greater than or equal to 20000.

4.8.3 Problem: Problem: Monte Carlo Simulation

Monte Carlo simulation uses random numbers and probability to solve problems. This method has a wide range of applications in computational mathematics, physics, chemistry, and finance. This section gives an example of using Monte Carlo simulation for estimating π.

To estimate π using the Monte Carlo method, draw a circle with its bounding square as shown below.



Assume the radius of the circle is `1`. So, the circle area is π and the square area is `4`. Randomly generate a point in the square. The probability for the point to fall in the circle is `circleArea / squareArea` = π / `4`.

Write a program that randomly generates `1000000` points in the square and let `numberOfHits` denote the number of points that fall in the circle. So, `numberOfHits` is approximately `1000000 * (`π `/ 4)`. π can be approximated as `4 * numberOfHits / 1000000`. The complete program is shown in Listing 4.10.

LISTING 4.10 MonteCarloSimulation.java

```
1 public class MonteCarloSimulation {
2   public static void main(String[] args) {
3     final int NUMBER_OF_TRIALS = 10000000;
4     int numberOfHits = 0;
5
6     for (int i = 0; i < NUMBER_OF_TRIALS; i++) {
7       double x = Math.random() * 2.0 - 1;
8       double y = Math.random() * 2.0 - 1;
9       if (x * x + y * y <= 1)
10        numberOfHits++;
11    }
12
13    double pi = 4.0 * numberOfHits / NUMBER_OF_TRIALS;
```

generate random points

check inside circle

estimate pi

```
14     System.out.println("PI is " + pi);
15   }
16 }
```

```
PI is 3.14124
```



The program repeatedly generates a random point (**x**, **y**) in the square in lines 7–8:

```
double x = Math.random() * 2.0 - 1;
double y = Math.random() * 2.0 - 1;
```

If $x^2 + y^2 \leq 1$, the point is inside the circle and **numberOfHits** is incremented by **1**. π is approximately **4 * numberOfHits / NUMBER_OF_TRIALS** (line 13).

4.9 Keywords *break* and *continue*



Pedagogical Note

Two keywords, **break** and **continue**, can be used in loop statements to provide additional controls. Using **break** and **continue** can simplify programming in some cases. Overusing or improperly using them, however, can make programs difficult to read and debug. (*Note to instructors*: You may skip this section without affecting the rest of the book.)

You have used the keyword **break** in a **switch** statement. You can also use **break** in a loop to immediately terminate the loop. Listing 4.11 presents a program to demonstrate the effect of using **break** in a loop.

break

LISTING 4.11 TestBreak.java

```
1 public class TestBreak {
2   public static void main(String[] args) {
3     int sum = 0;
4     int number = 0;
5
6     while (number < 20) {
7       number++;
8       sum += number;
9       if (sum >= 100)
10        break;
11    }
12
13    System.out.println("The number is " + number);
14    System.out.println("The sum is " + sum);
15  }
16 }
```

break

```
The number is 14
The sum is 105
```



The program in Listing 4.11 adds integers from **1** to **20** in this order to **sum** until **sum** is greater than or equal to **100**. Without the **if** statement (line 9), the program calculates the sum of the numbers from **1** to **20**. But with the **if** statement, the loop terminates when **sum** becomes greater than or equal to **100**. Without the **if** statement, the output would be:



```
The number is 20
The sum is 210
```

continue

You can also use the **continue** keyword in a loop. When it is encountered, it ends the current iteration. Program control goes to the end of the loop body. In other words, **continue** breaks out of an iteration while the **break** keyword breaks out of a loop. Listing 4.12 presents a program to demonstrate the effect of using **continue** in a loop.

LISTING 4.12 TestContinue.java

```
1 public class TestContinue {
2   public static void main(String[] args) {
3     int sum = 0;
4     int number = 0;
5
6     while (number < 20) {
7       number++;
8       if (number == 10 || number == 11)
9         continue;
10      sum += number;
11    }
12
13    System.out.println("The sum is " + sum);
14  }
15 }
```

continue



```
The sum is 189
```

The program in Listing 4.12 adds integers from **1** to **20** except **10** and **11** to **sum**. With the **if** statement in the program (line 8), the **continue** statement is executed when **number** becomes **10** or **11**. The **continue** statement ends the current iteration so that the rest of the statement in the loop body is not executed; therefore, **number** is not added to **sum** when it is **10** or **11**. Without the **if** statement in the program, the output would be as follows:



```
The sum is 210
```

In this case, all of the numbers are added to **sum**, even when **number** is **10** or **11**. Therefore, the result is **210**, which is **21** more than it was with the **if** statement.



Note

The **continue** statement is always inside a loop. In the **while** and **do-while** loops, the **loop-continuation-condition** is evaluated immediately after the **continue** statement. In the **for** loop, the **action-after-each-iteration** is performed, then the **loop-continuation-condition** is evaluated, immediately after the **continue** statement.

You can always write a program without using **break** or **continue** in a loop. See Review Question 4.18. In general, using **break** and **continue** is appropriate only if it simplifies coding and makes programs easier to read.

Listing 4.2 gives a program for guessing a number. You can rewrite it using a **break** statement, as shown in Listing 4.13.

LISTING 4.13 GuessNumberUsingBreak.java

```
1 import java.util.Scanner;
2
3 public class GuessNumberUsingBreak {
4   public static void main(String[] args) {
5     // Generate a random number to be guessed
6     int number = (int)(Math.random() * 101);                    generate a number
7
8     Scanner input = new Scanner(System.in);
9     System.out.println("Guess a magic number between 0 and 100");
10
11    while (true) {                                              loop continuously
12      // Prompt the user to guess the number
13      System.out.print("\nEnter your guess: ");                 enter a guess
14      int guess = input.nextInt();
15
16      if (guess == number) {
17        System.out.println("Yes, the number is " + number);
18        break;                                                  break
19      }
20      else if (guess > number)
21        System.out.println("Your guess is too high");
22      else
23        System.out.println("Your guess is too low");
24    } // End of loop
25  }
26 }
```

Using the **break** statement makes this program simpler and easier to read. However, you should use **break** and **continue** with caution. Too many **break** and **continue** statements will produce a loop with many exit points and make the program difficult to read.



Note

Some programming languages have a **goto** statement. The **goto** statement indiscriminately transfers control to any statement in the program and executes it. This makes your program vulnerable to errors. The **break** and **continue** statements in Java are different from **goto** statements. They operate only in a loop or a **switch** statement. The **break** statement breaks out of the loop, and the **continue** statement breaks out of the current iteration in the loop.

goto

4.9.1 Problem: Displaying Prime Numbers

An integer greater than **1** is *prime* if its only positive divisor is **1** or itself. For example, **2**, **3**, **5**, and **7** are prime numbers, but **4**, **6**, **8**, and **9** are not.

The problem is to display the first 50 prime numbers in five lines, each of which contains ten numbers. The problem can be broken into the following tasks:

- Determine whether a given number is prime.
- For **number** = **2**, **3**, **4**, **5**, **6**, . . . , test whether it is prime.
- Count the prime numbers.
- Print each prime number, and print ten numbers per line.

Obviously, you need to write a loop and repeatedly test whether a new **number** is prime. If the **number** is prime, increase the count by **1**. The **count** is **0** initially. When it reaches **50**, the loop terminates.

Here is the algorithm for the problem:

```
Set the number of prime numbers to be printed as
  a constant NUMBER_OF_PRIMES;
Use count to track the number of prime numbers and
  set an initial count to 0;
Set an initial number to 2;

while (count < NUMBER_OF_PRIMES) {
  Test whether number is prime;

  if number is prime {
    Print the prime number and increase the count;
  }

  Increment number by 1;
}
```

To test whether a number is prime, check whether it is divisible by 2, 3, 4, up to number/2. If a divisor is found, the number is not a prime. The algorithm can be described as follows:

```
Use a boolean variable isPrime to denote whether
  the number is prime; Set isPrime to true initially;

for (int divisor = 2; divisor <= number / 2; divisor++) {
  if (number % divisor == 0) {
    Set isPrime to false
    Exit the loop;
  }
}
```

The complete program is given in Listing 4.14.

LISTING 4.14 PrimeNumber.java

```
1  public class PrimeNumber {
2    public static void main(String[] args) {
3      final int NUMBER_OF_PRIMES = 50; // Number of primes to display
4      final int NUMBER_OF_PRIMES_PER_LINE = 10; // Display 10 per line
5      int count = 0; // Count the number of prime numbers
6      int number = 2; // A number to be tested for primeness
7
8      System.out.println("The first 50 prime numbers are \n");
9
10     // Repeatedly find prime numbers
11     while (count < NUMBER_OF_PRIMES) {
12       // Assume the number is prime
13       boolean isPrime = true; // Is the current number prime?
14
15       // Test whether number is prime
16       for (int divisor = 2; divisor <= number / 2; divisor++) {
17         if (number % divisor == 0) { // If true, number is not prime
18           isPrime = false; // Set isPrime to false
19           break; // Exit the for loop
20         }
21       }
22
23       // Print the prime number and increase the count
24       if (isPrime) {
25         count++; // Increase the count
```

count prime numbers

check primeness

exit loop

print if prime

```
26
27          if (count % NUMBER_OF_PRIMES_PER_LINE == 0) {
28            // Print the number and advance to the new line
29            System.out.println(number);
30          }
31          else
32            System.out.print(number + " ");
33        }
34
35        // Check if the next number is prime
36        number++;
37      }
38    }
39 }
```

```
The first 50 prime numbers are

2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
```



This is a complex program for novice programmers. The key to developing a programmatic solution to this problem, and to many other problems, is to break it into subproblems and develop solutions for each of them in turn. Do not attempt to develop a complete solution in the first trial. Instead, begin by writing the code to determine whether a given number is prime, then expand the program to test whether other numbers are prime in a loop.

subproblem

To determine whether a number is prime, check whether it is divisible by a number between 2 and `number/2` inclusive (line 16). If so, it is not a prime number (line 18); otherwise, it is a prime number. For a prime number, display it. If the count is divisible by 10 (lines 27–30), advance to a new line. The program ends when the count reaches 50.

The program uses the `break` statement in line 19 to exit the `for` loop as soon as the number is found to be a nonprime. You can rewrite the loop (lines 16–21) without using the `break` statement, as follows:

```
for (int divisor = 2; divisor <= number / 2 && isPrime;
     divisor++) {
  // If true, the number is not prime
  if (number % divisor == 0) {
    // Set isPrime to false, if the number is not prime
    isPrime = false;
  }
}
```

However, using the `break` statement makes the program simpler and easier to read in this case.

4.10 (GUI) Controlling a Loop with a Confirmation Dialog

A sentinel-controlled loop can be implemented using a confirmation dialog. The answers *Yes* or *No* continue or terminate the loop. The template of the loop may look as follows:

confirmation dialog

```
int option = JOptionPane.YES_OPTION;
while (option == JOptionPane.YES_OPTION) {
  System.out.println("continue loop");
  option = JOptionPane.showConfirmDialog(null, "Continue?");
}
```

Listing 4.15 rewrites Listing 4.4, SentinelValue.java, using a confirmation dialog box. A sample run is shown in Figure 4.4.

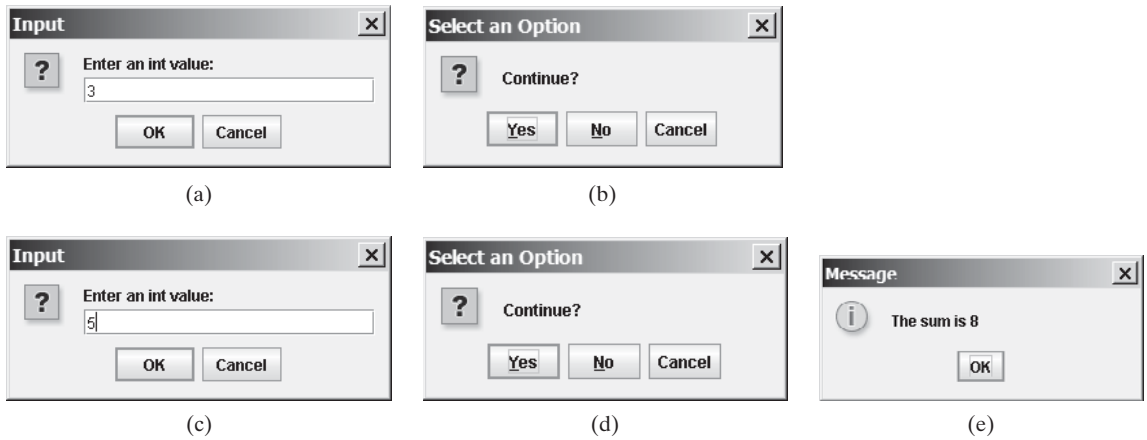


FIGURE 4.4 The user enters 3 in (a), clicks Yes in (b), enters 5 in (c), clicks No in (d), and the result is shown in (e).

LISTING 4.15 SentinelValueUsingConfirmationDialog.java

```
1  import javax.swing.JOptionPane;
2
3  public class SentinelValueUsingConfirmationDialog {
4    public static void main(String[] args) {
5      int sum = 0;
6
7      // Keep reading data until the user answers No
8      int option = JOptionPane.YES_OPTION;
9      while (option == JOptionPane.YES_OPTION) {
10       // Read the next data
11       String dataString = JOptionPane.showInputDialog(
12         "Enter an int value: ");
13       int data = Integer.parseInt(dataString);
14
15       sum += data;
16
17       option = JOptionPane.showConfirmDialog(null, "Continue?");
18     }
19
20     JOptionPane.showMessageDialog(null, "The sum is " + sum);
21   }
22 }
```

confirmation option
check option
input dialog
confirmation dialog
message dialog

A program displays an input dialog to prompt the user to enter an integer (line 11) and adds it to `sum` (line 15). Line 17 displays a confirmation dialog to let the user decide whether to continue the input. If the user clicks *Yes*, the loop continues; otherwise the loop exits. Finally the program displays the result in a message dialog box (line 20).

KEY TERMS

`break` statement 136
`continue` statement 136
`do-while` loop 124
`for` loop 126
loop control structure 127
infinite loop 117

input redirection 124
iteration 116
labeled continue statement 136
loop 116
`loop-continuation-condition` 116
loop body 116
nested loop 129
off-by-one error 124
output redirection 124
sentinel value 122
`while` loop 116

Chapter Summary

1. There are three types of repetition statements: the `while` loop, the `do-while` loop, and the `for` loop.

2. The part of the loop that contains the statements to be repeated is called the *loop body*.

3. A one-time execution of a loop body is referred to as an *iteration of the loop*.

4. An infinite loop is a loop statement that executes infinitely.

5. In designing loops, you need to consider both the loop control structure and the loop body.

6. The `while` loop checks the `loop-continuation-condition` first. If the condition is `true`, the loop body is executed; if it is `false`, the loop terminates.

7. The `do-while` loop is similar to the `while` loop, except that the `do-while` loop executes the loop body first and then checks the `loop-continuation-condition` to decide whether to continue or to terminate.

8. Since the `while` loop and the `do-while` loop contain the `loop-continuation-condition`, which is dependent on the loop body, the number of repetitions is determined by the loop body. The `while` loop and the `do-while` loop often are used when the number of repetitions is unspecified.

9. A *sentinel value* is a special value that signifies the end of the loop.

10. The `for` loop generally is used to execute a loop body a predictable number of times; this number is not determined by the loop body.

11. The `for` loop control has three parts. The first part is an initial action that often initializes a control variable. The second part, the loop-continuation-condition, determines whether the loop body is to be executed. The third part is executed after each iteration and is often used to adjust the control variable. Usually, the loop control variables are initialized and changed in the control structure.

12. The `while` loop and `for` loop are called *pretest loops* because the continuation condition is checked before the loop body is executed.

13. The `do-while` loop is called *posttest loop* because the condition is checked after the loop body is executed.

14. Two keywords, `break` and `continue`, can be used in a loop.

15. The `break` keyword immediately ends the innermost loop, which contains the break.

16. The `continue` keyword only ends the current iteration.

Review Questions

Sections 4.2–4.4

4.1 Analyze the following code. Is `count < 100` always `true`, always `false`, or sometimes `true` or sometimes `false` at Point A, Point B, and Point C?

```
int count = 0;
while (count < 100) {
  // Point A
  System.out.println("Welcome to Java!\n");
  count++;
  // Point B
}
  // Point C
```

4.2 What is wrong if `guess` is initialized to `0` in line 11 in Listing 4.2?

4.3 How many times is the following loop body repeated? What is the printout of the loop?

```
int i = 1;
while (i < 10)
  if (i % 2 == 0)
    System.out.println(i);
```

(a)

```
int i = 1;
while (i < 10)
  if (i % 2 == 0)
    System.out.println(i++);
```

(b)

```
int i = 1;
while (i < 10)
  if ((i++) % 2 == 0)
    System.out.println(i);
```

(c)

4.4 What are the differences between a `while` loop and a `do-while` loop? Convert the following `while` loop into a `do-while` loop.

```
int sum = 0;
int number = input.nextInt();
while (number != 0) {
  sum += number;
  number = input.nextInt();
}
```

4.5 Do the following two loops result in the same value in `sum`?

```
for (int i = 0; i < 10; ++i) {
  sum += i;
}
```

(a)

```
for (int i = 0; i < 10; i++) {
  sum += i;
}
```

(b)

4.6 What are the three parts of a `for` loop control? Write a `for` loop that prints the numbers from `1` to `100`.

4.7 Suppose the input is `2 3 4 5 0`. What is the output of the following code?

```
import java.util.Scanner;

public class Test {
```

```
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    int number, max;
    number = input.nextInt();
    max = number;

    while (number != 0) {
      number = input.nextInt();
      if (number > max)
        max = number;
    }

    System.out.println("max is " + max);
    System.out.println("number " + number);
  }
}
```

4.8 Suppose the input is 2 3 4 5 0. What is the output of the following code?

```
import java.util.Scanner;

public class Test {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    int number, sum = 0, count;

    for (count = 0; count < 5; count++) {
      number = input.nextInt();
      sum += number;
    }

    System.out.println("sum is " + sum);
    System.out.println("count is " + count);
  }
}
```

4.9 Suppose the input is 2 3 4 5 0. What is the output of the following code?

```
import java.util.Scanner;

public class Test {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    int number, max;
    number = input.nextInt();
    max = number;

    do {
      number = input.nextInt();
      if (number > max)
        max = number;
    } while (number != 0);

    System.out.println("max is " + max);
    System.out.println("number " + number);
  }
}
```

4.10 What does the following statement do?

```
for ( ; ; ) {
  do something;
}
```

4.11 If a variable is declared in the `for` loop control, can it be used after the loop exits?

4.12 Can you convert a `for` loop to a `while` loop? List the advantages of using `for` loops.

4.13 Convert the following `for` loop statement to a `while` loop and to a `do-while` loop:

```
long sum = 0;
for (int i = 0; i <= 1000; i++)
  sum = sum + i;
```

4.14 Will the program work if `n1` and `n2` are replaced by `n1 / 2` and `n2 / 2` in line 17 in Listing 4.8?

Section 4.9

4.15 What is the keyword `break` for? What is the keyword `continue` for? Will the following program terminate? If so, give the output.

```
int balance = 1000;
while (true) {
  if (balance < 9)
    break;
  balance = balance - 9;
}

System.out.println("Balance is "
  + balance);
```

(a)

```
int balance = 1000;
while (true) {
  if (balance < 9)
    continue;
  balance = balance - 9;
}

System.out.println("Balance is "
  + balance);
```

(b)

4.16 Can you always convert a `while` loop into a `for` loop? Convert the following `while` loop into a `for` loop.

```
int i = 1;
int sum = 0;
while (sum < 10000) {
  sum = sum + i;
  i++;
}
```

4.17 The `for` loop on the left is converted into the `while` loop on the right. What is wrong? Correct it.

```
for (int i = 0; i < 4; i++) {
  if (i % 3 == 0) continue;
  sum += i;
}
```

Converted

Wrong conversion

```
int i = 0;
while (i < 4) {
  if(i % 3 == 0) continue;
  sum += i;
  i++;
}
```

4.18 Rewrite the programs `TestBreak` and `TestContinue` in Listings 4.11 and 4.12 without using `break` and `continue`.

4.19 After the `break` statement is executed in the following loop, which statement is executed? Show the output.

```
for (int i = 1; i < 4; i++) {
  for (int j = 1; j < 4; j++) {
    if (i * j > 2)
      break;

    System.out.println(i * j);
  }

  System.out.println(i);
}
```

4.20 After the `continue` statement is executed in the following loop, which statement is executed? Show the output.

```
for (int i = 1; i < 4; i++) {
  for (int j = 1; j < 4; j++) {
    if (i * j > 2)
      continue;

    System.out.println(i * j);
  }

  System.out.println(i);
}
```

Comprehensive

4.21 Identify and fix the errors in the following code:

```
 1 public class Test {
 2   public void main(String[] args) {
 3     for (int i = 0; i < 10; i++);
 4       sum += i;
 5
 6     if (i < j);
 7       System.out.println(i)
 8     else
 9       System.out.println(j);
10
11     while (j < 10);
12     {
13       j++;
14     };
15
16     do {
17       j++;
18     } while (j < 10)
19   }
20 }
```

4.22 What is wrong with the following programs?

```
1 public class ShowErrors {
2   public static void main(String[] args) {
3     int i;
4     int j = 5;
5
6     if (j > 3)
7       System.out.println(i + 4);
8   }
9 }
```

(a)

```
1 public class ShowErrors {
2   public static void main(String[] args) {
3     for (int i = 0; i < 10; i++);
4       System.out.println(i + 4);
5   }
6 }
```

(b)

4.23 Show the output of the following programs. (*Tip*: Draw a table and list the variables in the columns to trace these programs.)

```
public class Test {
  /** Main method */
  public static void main(String[] args) {
    for (int i = 1; i < 5; i++) {
      int j = 0;
      while (j < i) {
        System.out.print(j + " ");
        j++;
      }
    }
  }
}
```

(a)

```
public class Test {
  /** Main method */
  public static void main(String[] args) {
    int i = 0;
    while (i < 5) {
      for (int j = i; j > 1; j-)
        System.out.print(j + " ");
      System.out.println("****");
      i++;
    }
  }
}
```

(b)

```
public class Test {
  public static void main(String[] args) {
    int i = 5;
    while (i >= 1) {
      int num = 1;
      for (int j = 1; j <= i; j++) {
        System.out.print(num + "xxx");
        num *= 2;
      }

      System.out.println();
      i--;
    }
  }
}
```

(c)

```
public class Test {
  public static void main(String[] args) {
    int i = 1;
    do {
      int num = 1;
      for (int j = 1; j <= i; j++) {
        System.out.print(num + "G");
        num += 2;
      }

      System.out.println();
      i++;
    } while (i <= 5);
  }
}
```

(d)

4.24 What is the output of the following program? Explain the reason.

```
int x = 80000000;

while (x > 0)
  x++;

System.out.println("x is " + x);
```

4.25 Count the number of iterations in the following loops.

```
int count = 0;
while (count < n) {
  count++;
}
```

(a)

```
for (int count = 0;
   count <= n; count++) {
}
```

(b)

```
int count = 5;
while (count < n) {
  count++;
}
```

(c)

```
int count = 5;
while (count < n) {
  count = count + 3;
}
```

(d)

Programming Exercises



Pedagogical Note

For each problem, read it several times until you understand the problem. Think how to solve the problem before coding. Translate your logic into a program.

read and think before coding

A problem often can be solved in many different ways. Students are encouraged to explore various solutions.

explore solutions

Sections 4.2–4.7

4.1* (*Counting positive and negative numbers and computing the average of numbers*) Write a program that reads an unspecified number of integers, determines how many positive and negative values have been read, and computes the total and average of the input values (not counting zeros). Your program ends with the input `0`. Display the average as a floating-point number. Here is a sample run:



```
Enter an int value, the program exits if the input is 0:
  1 2 -1 3 0 Enter
The number of positives is 3
The number of negatives is 1
The total is 5
The average is 1.25
```

4.2 (*Repeating additions*) Listing 4.3, SubtractionQuizLoop.java, generates five random subtraction questions. Revise the program to generate ten random addition questions for two integers between `1` and `15`. Display the correct count and test time.

4.3 (*Conversion from kilograms to pounds*) Write a program that displays the following table (note that `1` kilogram is `2.2` pounds):

```
Kilograms       Pounds

1                 2.2
3                 6.6
...
197             433.4
199             437.8
```

4.4 (*Conversion from miles to kilometers*) Write a program that displays the following table (note that 1 mile is 1.609 kilometers):

```
Miles       Kilometers

1           1.609
2           3.218
...
9           14.481
10          16.090
```

4.5 (*Conversion from kilograms to pounds*) Write a program that displays the following two tables side by side (note that 1 kilogram is 2.2 pounds):

```
Kilograms     Pounds          Pounds          Kilograms

1             2.2             20              9.09
3             6.6             25              11.36
...
197           433.4           510             231.82
199           437.8           515             234.09
```

4.6 (*Conversion from miles to kilometers*) Write a program that displays the following two tables side by side (note that 1 mile is 1.609 kilometers):

```
Miles       Kilometers      Kilometers      Miles

1           1.609           20              12.430
2           3.218           25              15.538
...
9           14.481          60              37.290
10          16.090          65              40.398
```

4.7** (*Financial application: computing future tuition*) Suppose that the tuition for a university is $10,000 this year and increases 5% every year. Write a program that computes the tuition in ten years and the total cost of four years' worth of tuition starting ten years from now.

4.8 (*Finding the highest score*) Write a program that prompts the user to enter the number of students and each student's name and score, and finally displays the name of the student with the highest score.

4.9* (*Finding the two highest scores*) Write a program that prompts the user to enter the number of students and each student's name and score, and finally displays the student with the highest score and the student with the second-highest score.

4.10 (*Finding numbers divisible by 5 and 6*) Write a program that displays all the numbers from 100 to 1000, ten per line, that are divisible by 5 and 6.

4.11 (*Finding numbers divisible by 5 or 6, but not both*) Write a program that displays all the numbers from 100 to 200, ten per line, that are divisible by 5 or 6, but not both.

4.12 (*Finding the smallest n such that $n^2 > 12{,}000$*) Use a `while` loop to find the smallest integer `n` such that n^2 is greater than 12,000.

4.13 (*Finding the largest n such that $n^3 < 12{,}000$*) *Use a* `while` loop to find the largest integer `n` such that n^3 is less than 12,000.

4.14* (*Displaying the ACSII character table*) Write a program that prints the characters in the ASCII character table from `'!'` to `'~'`. Print ten characters per line. The ASCII table is shown in Appendix B.

Section 4.8

4.15* (*Computing the greatest common divisor*) Another solution for Listing 4.8 to find the greatest common divisor of two integers `n1` and `n2` is as follows: First find `d` to be the minimum of `n1` and `n2`, then check whether `d`, `d-1`, `d-2`, ..., `2`, or `1` is a divisor for both `n1` and `n2` in this order. The first such common divisor is the greatest common divisor for `n1` and `n2`. Write a program that prompts the user to enter two positive integers and displays the gcd.

4.16** (*Finding the factors of an integer*) Write a program that reads an integer and displays all its smallest factors in increasing order. For example, if the input integer is `120`, the output should be as follows: `2`, `2`, `2`, `3`, `5`.

4.17** (*Displaying pyramid*) Write a program that prompts the user to enter an integer from `1` to `15` and displays a pyramid, as shown in the following sample run:

```
Enter the number of lines: 7 Enter
                    1
                  2 1 2
                3 2 1 2 3
              4 3 2 1 2 3 4
            5 4 3 2 1 2 3 4 5
          6 5 4 3 2 1 2 3 4 5 6
        7 6 5 4 3 2 1 2 3 4 5 6 7
```



4.18* (*Printing four patterns using loops*) Use nested loops that print the following patterns in four separate programs:

```
Pattern I        Pattern II       Pattern III      Pattern IV
1                1 2 3 4 5 6              1       1 2 3 4 5 6
1 2              1 2 3 4 5              2 1         1 2 3 4 5
1 2 3            1 2 3 4              3 2 1           1 2 3 4
1 2 3 4          1 2 3              4 3 2 1             1 2 3
1 2 3 4 5        1 2              5 4 3 2 1               1 2
1 2 3 4 5 6      1              6 5 4 3 2 1                 1
```

4.19** (*Printing numbers in a pyramid pattern*) Write a nested `for` loop that prints the following output:

```
                    1
                  1 2 1
                1 2 4 2 1
              1 2 4 8 4 2 1
            1 2 4 8 16 8 4 2 1
          1 2 4 8 16 32 16 8 4 2 1
        1 2 4 8 16 32 64 32 16 8 4 2 1
      1 2 4 8 16 32 64 128 64 32 16 8 4 2 1
```

4.20* (*Printing prime numbers between 2 and 1000*) Modify Listing 4.14 to print all the prime numbers between 2 and 1000, inclusive. Display eight prime numbers per line.

Comprehensive

4.21** (*Financial application: comparing loans with various interest rates*) Write a program that lets the user enter the loan amount and loan period in number of years

and displays the monthly and total payments for each interest rate starting from 5% to 8%, with an increment of 1/8. Here is a sample run:



```
Loan Amount: 10000 Enter
Number of Years: 5 Enter
Interest Rate               Monthly Payment    Total Payment

5%                          188.71             11322.74
5.125%                      189.28             11357.13
5.25%                       189.85             11391.59
...
7.875%                      202.17             12129.97
8.0%                        202.76             12165.83
```

For the formula to compute monthly payment, see Listing 2.8, ComputeLoan.java.



Video Note
Display loan schedule

4.22** (*Financial application: loan amortization schedule*) The monthly payment for a given loan pays the principal and the interest. The monthly interest is computed by multiplying the monthly interest rate and the balance (the remaining principal). The principal paid for the month is therefore the monthly payment minus the monthly interest. Write a program that lets the user enter the loan amount, number of years, and interest rate and displays the amortization schedule for the loan. Here is a sample run:



```
Loan Amount: 10000
Number of Years: 1
Annual Interest Rate: 7%

Monthly Payment: 865.26
Total Payment: 10383.21

Payment#          Interest          Principal         Balance

1                 58.33             806.93            9193.07
2                 53.62             811.64            8381.43
...
11                10.0              855.26            860.27
12                5.01              860.25            0.01
```



Note

The balance after the last payment may not be zero. If so, the last payment should be the normal monthly payment plus the final balance.

Hint: Write a loop to print the table. Since the monthly payment is the same for each month, it should be computed before the loop. The balance is initially the loan amount. For each iteration in the loop, compute the interest and principal, and update the balance. The loop may look like this:

```
for (i = 1; i <= numberOfYears * 12; i++) {
  interest = monthlyInterestRate * balance;
  principal = monthlyPayment - interest;
  balance = balance - principal;
  System.out.println(i + "\t\t" + interest
    + "\t\t" + principal + "\t\t" + balance);
}
```

4.23* (*Obtaining more accurate results*) In computing the following series, you will obtain more accurate results by computing from right to left rather than from left to right:

$$1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

Write a program that compares the results of the summation of the preceding series, computing from left to right and from right to left with `n` = `50000`.

4.24* (*Summing a series*) Write a program to sum the following series:

$$\frac{1}{3} + \frac{3}{5} + \frac{5}{7} + \frac{7}{9} + \frac{9}{11} + \frac{11}{13} + \ldots + \frac{95}{97} + \frac{97}{99}$$

4.25** (*Computing π*) You can approximate π by using the following series:

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \ldots + \frac{1}{2i-1} - \frac{1}{2i+1}\right)$$

Write a program that displays the π value for `i` = `10000`, `20000`, ..., and `100000`.

4.26** (*Computing e*) You can approximate `e` using the following series:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots + \frac{1}{i!}$$

Write a program that displays the `e` value for `i` = `10000`, `20000`, ..., and `100000`.

(*Hint*: Since $i! = i \times (i-1) \times \ldots \times 2 \times 1$, then $\frac{1}{i!}$ is $\frac{1}{i(i-1)!}$

Initialize `e` and `item` to be `1` and keep adding a new `item` to `e`. The new item is the previous item divided by `i` for i = `2`, `3`, `4`, ...)

4.27** (*Displaying leap years*) Write a program that displays all the leap years, ten per line, in the twenty-first century (from 2001 to 2100).

4.28** (*Displaying the first days of each month*) Write a program that prompts the user to enter the year and first day of the year, and displays the first day of each month in the year on the console. For example, if the user entered the year `2005`, and `6` for Saturday, January 1, 2005, your program should display the following output (note that Sunday is `0`):

```
January 1, 2005 is Saturday
...
December 1, 2005 is Thursday
```

4.29** (*Displaying calendars*) Write a program that prompts the user to enter the year and first day of the year and displays the calendar table for the year on the console. For example, if the user entered the year `2005`, and `6` for Saturday, January 1, 2005, your program should display the calendar for each month in the year, as follows:

January 2005

Sun	Mon	Tue	Wed	Thu	Fri	Sat
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

...

December 2005

Sun	Mon	Tue	Wed	Thu	Fri	Sat
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

4.30* (*Financial application: compound value*) Suppose you save **$100** *each* month into a savings account with the annual interest rate **5%**. So, the monthly interest rate is **0.05 / 12 = 0.00417**. After the first month, the value in the account becomes

```
100 * (1 + 0.00417) = 100.417
```

After the second month, the value in the account becomes

```
(100 + 100.417) * (1 + 0.00417) = 201.252
```

After the third month, the value in the account becomes

```
(100 + 201.252) * (1 + 0.00417) = 302.507
```

and so on.

Write a program that prompts the user to enter an amount (e.g., **100**), the annual interest rate (e.g., **5**), and the number of months (e.g., **6**) and displays the amount in the savings account after the given month.

4.31* (*Financial application: computing CD value*) Suppose you put $10,000 into a CD with an annual percentage yield of 5.75%. After one month, the CD is worth

```
10000 + 10000 * 5.75 / 1200 = 10047.91
```

After two months, the CD is worth

```
10047.91 + 10047.91 * 5.75 / 1200 = 10096.06
```

After three months, the CD is worth

```
10096.06 + 10096.06 * 5.75 / 1200 = 10144.43
```

and so on.

Write a program that prompts the user to enter an amount (e.g., **10000**), the annual percentage yield (e.g., **5.75**), and the number of months (e.g., **18**) and displays a table as shown in the sample run.



```
Enter the initial deposit amount: 10000 Enter
Enter annual percentage yield: 5.75 Enter
Enter maturity period (number of months): 18 Enter

Month               CD Value
1                   10047.91
2                   10096.06
...
17                  10846.56
18                  10898.54
```

4.32** (*Game: lottery*) Revise Listing 3.9, Lottery.java, to generate a lottery of a two-digit number. The two digits in the number are distinct.

(*Hint*: Generate the first digit. Use a loop to continuously generate the second digit until it is different from the first digit.)

4.33** (*Perfect number*) A positive integer is called a *perfect number* if it is equal to the sum of all of its positive divisors, excluding itself. For example, `6` is the first perfect number because `6 = 3 + 2 + 1`. The next is `28 = 14 + 7 + 4 + 2 + 1`. There are four perfect numbers less than `10000`. Write a program to find all these four numbers.

4.34*** (*Game: scissor, rock, paper*) Exercise 3.17 gives a program that plays the scissor-rock-paper game. Revise the program to let the user continuously play until either the user or the computer wins more than two times.

4.35* (*Summation*) Write a program that computes the following summation.

$$\frac{1}{1+\sqrt{2}}+\frac{1}{\sqrt{2}+\sqrt{3}}+\frac{1}{\sqrt{3}+\sqrt{4}}+\ldots+\frac{1}{\sqrt{624}+\sqrt{625}}$$

4.36** (*Business application: checking ISBN*) Use loops to simplify Exercise 3.19.

4.37** (*Decimal to binary*) Write a program that prompts the user to enter a decimal integer and displays its corresponding binary value. Don't use Java's `Integer.toBinaryString(int)` in this program.

4.38** (*Decimal to hex*) Write a program that prompts the user to enter a decimal integer and displays its corresponding hexadecimal value. Don't use Java's `Integer.toHexString(int)` in this program.

4.39* (*Financial application: finding the sales amount*) You have just started a sales job in a department store. Your pay consists of a base salary and a commission. The base salary is $5,000. The scheme shown below is used to determine the commission rate.

Sales Amount	Commission Rate
$0.01–$5,000	8 percent
$5,000.01–$10,000	10 percent
$10,000.01 and above	12 percent

Your goal is to earn $30,000 a year. Write a program that finds out the minimum amount of sales you have to generate in order to make $30,000.

4.40 (*Simulation: head or tail*) Write a program that simulates flipping a coin one million times and displays the number of heads and tails.

4.41** (*Occurrence of max numbers*) Write a program that reads integers, finds the largest of them, and counts its occurrences. Assume that the input ends with number `0`. Suppose that you entered `3 5 2 5 5 5 0`; the program finds that the largest is `5` and the occurrence count for `5` is `4`.

(*Hint*: Maintain two variables, `max` and `count`. `max` stores the current max number, and `count` stores its occurrences. Initially, assign the first number to `max` and `1` to `count`. Compare each subsequent number with `max`. If the number is greater than `max`, assign it to `max` and reset `count` to `1`. If the number is equal to `max`, increment `count` by `1`.)

```
Enter numbers: 3 5 2 5 5 5 0 Enter
The largest number is 5
The occurrence count of the largest number is 4
```



4.42* (*Financial application: finding the sales amount*) Rewrite Exercise 4.39 as follows:

- Use a `for` loop instead of a `do-while` loop.
- Let the user enter `COMMISSION_SOUGHT` instead of fixing it as a constant.

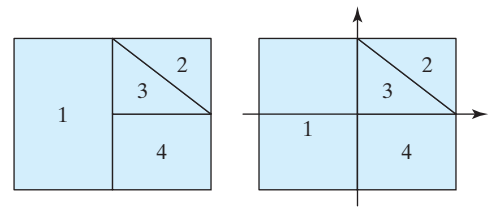
4.43* (*Simulation: clock countdown*) Write a program that prompts the user to enter the number of seconds, displays a message at every second, and terminates when the time expires. Here is a sample run:



```
Enter the number of second: 3 Enter
2 seconds remaining
1 second remaining
Stopped
```

4.44** (*Monte Carlo simulation*) A square is divided into four smaller regions as shown below in (a). If you throw a dart into the square 1000000 times, what is the probability for a dart to fall into an odd-numbered region? Write a program to simulate the process and display the result.

(*Hint*: Place the center of the square in the center of a coordinate system, as shown in (b). Randomly generate a point in the square and count the number of times a point falls into an odd-numbered region.)



(a) (b)

4.45* (*Math: combinations*) Write a program that displays all possible combinations for picking two numbers from integers `1` to `7`. Also display the total number of all combinations.



```
1 2
1 3
...
...
```

4.46* (*Computer architecture: bit-level operations*) A `short` value is stored in `16` bits. Write a program that prompts the user to enter a short integer and displays the `16` bits for the integer. Here are sample runs:



```
Enter an integer: 5 Enter
The bits are 0000000000000101
```



```
Enter an integer: -5 Enter
The bits are 1111111111111011
```

(*Hint*: You need to use the bitwise right shift operator (`>>`) and the bitwise AND operator (`&`), which are covered in Supplement III.D on the Companion Website.)

CHAPTER 5

METHODS

Objectives

- To define methods (§5.2).
- To invoke methods with a return value (§5.3).
- To invoke methods without a return value (§5.4).
- To pass arguments by value (§5.5).
- To develop reusable code that is modular, easy to read, easy to debug, and easy to maintain (§5.6).
- To write a method that converts decimals to hexadecimals (§5.7).
- To use method overloading and understand ambiguous overloading (§5.8).
- To determine the scope of variables (§5.9).
- To solve mathematics problems using the methods in the `Math` class (§§5.10–5.11).
- To apply the concept of method abstraction in software development (§5.12).
- To design and implement methods using stepwise refinement (§5.12).



5.1 Introduction

problem

Suppose that you need to find the sum of integers from 1 to 10, from 20 to 30, and from 35 to 45, respectively. You may write the code as follows:

```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

sum = 0;
for (int i = 20; i <= 30; i++)
  sum += i;
System.out.println("Sum from 20 to 30 is " + sum);

sum = 0;
for (int i = 35; i <= 45; i++)
  sum += i;
System.out.println("Sum from 35 to 45 is " + sum);
```

You may have observed that computing sum from 1 to 10, from 20 to 30, and from 35 to 45 are very similar except that the starting and ending integers are different. Wouldn't it be nice if we could write the common code once and reuse it without rewriting it? We can do so by defining a method. The method is for creating reusable code.

why methods?

The preceding code can be simplified as follows:

define **sum** method

```
1 public static int sum(int i1, int i2) {
2   int sum = 0;
3   for (int i = i1; i <= i2; i++)
4     sum += i;
5
6   return sum;
7 }
8
```

main method
invoke **sum**

```
9  public static void main(String[] args) {
10   System.out.println("Sum from 1 to 10 is " + sum(1, 10));
11   System.out.println("Sum from 20 to 30 is " + sum(20, 30));
12   System.out.println("Sum from 35 to 45 is " + sum(35, 45));
13 }
```

Lines 1–7 define the method named `sum` with two parameters `i` and `j`. The statements in the `main` method invoke `sum(1, 10)` to compute the sum from 1 to 10, `sum(20, 30)` to compute the sum from 20 to 30, and `sum(35, 45)` to compute the sum from 35 to 45.

A method is a collection of statements grouped together to perform an operation. In earlier chapters you have used predefined methods such as `System.out.println`, `JOptionPane.showMessageDialog`, `JOptionPane.showInputDialog`, `Integer.parseInt`, `Double.parseDouble`, `System.exit`, `Math.pow`, and `Math.random`. These methods are defined in the Java library. In this chapter, you will learn how to define your own methods and apply method abstraction to solve complex problems.

5.2 Defining a Method

The syntax for defining a method is as follows:

```
modifier returnValueType methodName(list of parameters) {
  // Method body;
}
```

Let's look at a method created to find which of two integers is bigger. This method, named `max`, has two `int` parameters, `num1` and `num2`, the larger of which is returned by the method. Figure 5.1 illustrates the components of this method.

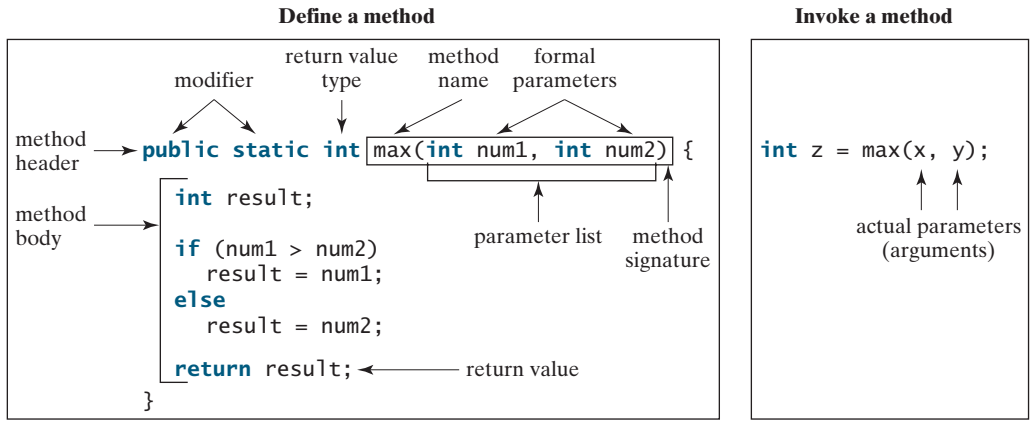


FIGURE 5.1 A method definition consists of a method header and a method body.

The *method header* specifies the *modifiers*, *return value type*, *method name*, and *parameters* of the method. The static modifier is used for all the methods in this chapter. The reason for using it will be discussed in Chapter 8, "Objects and Classes." — method header

A method may return a value. The `returnValueType` is the data type of the value the method returns. Some methods perform desired operations without returning a value. In this case, the `returnValueType` is the keyword `void`. For example, the `returnValueType` is `void` in the `main` method, as well as in `System.exit`, `System.out.println`, and `JOptionPane.showMessageDialog`. If a method returns a value, it is called a *value-returning method*, otherwise it is a *void method*. — value-returning method; void method

The variables defined in the method header are known as *formal parameters* or simply *parameters*. A parameter is like a placeholder. When a method is invoked, you pass a value to the parameter. This value is referred to as an *actual parameter or argument*. The *parameter list* refers to the type, order, and number of the parameters of a method. The method name and the parameter list together constitute the *method signature*. Parameters are optional; that is, a method may contain no parameters. For example, the `Math.random()` method has no parameters. — parameter; argument; parameter list; method signature

The method body contains a collection of statements that define what the method does. The method body of the `max` method uses an `if` statement to determine which number is larger and return the value of that number. In order for a value-returning method to return a result, a return statement using the keyword `return` is *required*. The method terminates when a return statement is executed.



Note

In certain other languages, methods are referred to as *procedures* and *functions*. A value-returning method is called a *function*; a void method is called a *procedure*.



Caution

In the method header, you need to declare a separate data type for each parameter. For instance, `max(int num1, int num2)` is correct, but `max(int num1, num2)` is wrong.



Note

We say "*define* a method" and "*declare* a variable." We are making a subtle distinction here. A definition defines what the defined item is, but a declaration usually involves allocating memory to store data for the declared item. — define vs. declare

5.3 Calling a Method

In creating a method, you define what the method is to do. To use a method, you have to *call* or *invoke* it. There are two ways to call a method, depending on whether the method returns a value or not.

If the method returns a value, a call to the method is usually treated as a value. For example,

```
int larger = max(3, 4);
```

calls `max(3, 4)` and assigns the result of the method to the variable `larger`. Another example of a call that is treated as a value is

```
System.out.println(max(3, 4));
```

which prints the return value of the method call `max(3, 4)`.

If the method returns `void`, a call to the method must be a statement. For example, the method `println` returns `void`. The following call is a statement:

```
System.out.println("Welcome to Java!");
```



Note

A value-returning method can also be invoked as a statement in Java. In this case, the caller simply ignores the return value. This is not often done but is permissible if the caller is not interested in the return value.

When a program calls a method, program control is transferred to the called method. A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached.

Listing 5.1 shows a complete program that is used to test the `max` method.



Video Note
Define/invoke max method

LISTING 5.1 TestMax.java

```
1  public class TestMax {
2    /** Main method */
3    public static void main(String[] args) {
4      int i = 5;
5      int j = 2;
6      int k = max(i, j);
7      System.out.println("The maximum between " + i +
8        " and " + j + " is " + k);
9    }
10
11   /** Return the max between two numbers */
12   public static int max(int num1, int num2) {
13     int result;
14
15     if (num1 > num2)
16       result = num1;
17     else
18       result = num2;
19
20     return result;
21   }
22 }
```

main method

invoke max

define method



```
The maximum between 5 and 2 is 5
```

	line#	i	j	k	num1	num2	result
	4	5					
	5		2				
Invoking max	12				5	2	
	13						undefined
	16						5
	6			5			

This program contains the `main` method and the `max` method. The `main` method is just like any other method except that it is invoked by the JVM. main method

The `main` method's header is always the same. Like the one in this example, it includes the modifiers `public` and `static`, return value type `void`, method name `main`, and a parameter of the `String[]` type. `String[]` indicates that the parameter is an array of `String`, a subject addressed in Chapter 6.

The statements in `main` may invoke other methods that are defined in the class that contains the `main` method or in other classes. In this example, the `main` method invokes `max(i, j)`, which is defined in the same class with the `main` method.

When the `max` method is invoked (line 6), variable `i`'s value `5` is passed to `num1`, and variable `j`'s value `2` is passed to `num2` in the `max` method. The flow of control transfers to the `max` method. The `max` method is executed. When the `return` statement in the `max` method is executed, the `max` method returns the control to its caller (in this case the caller is the `main` method). This process is illustrated in Figure 5.2. max method

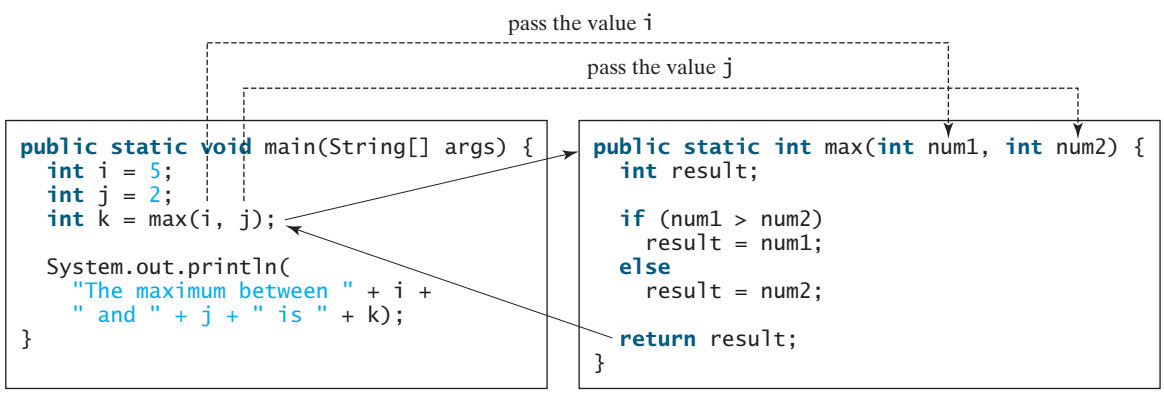


FIGURE 5.2 When the `max` method is invoked, the flow of control transfers to it. Once the `max` method is finished, it returns control back to the caller.



Caution

A `return` statement is required for a value-returning method. The method shown below in (a) is logically correct, but it has a compile error because the Java compiler thinks it possible that this method returns no value.

To fix this problem, delete `if (n < 0)` in (a), so that the compiler will see a `return` statement to be reached regardless of how the `if` statement is evaluated.

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else if (n < 0)
    return -1;
}
```

(a)

Should be

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else
    return -1;
}
```

(b)

reusing method

Note

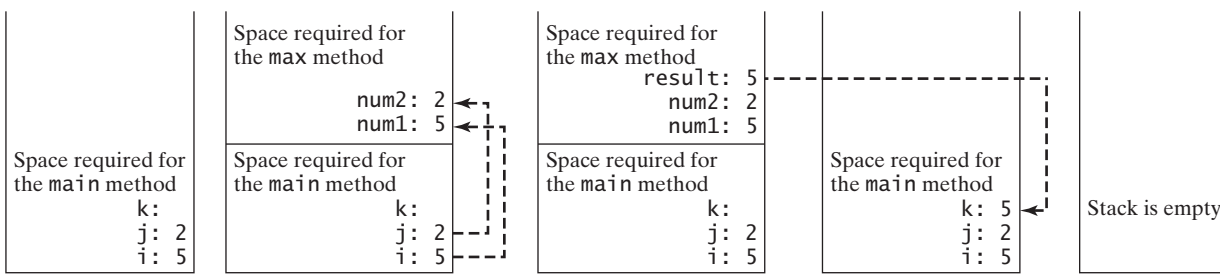
Methods enable code sharing and reuse. The `max` method can be invoked from any class besides `TestMax`. If you create a new class, you can invoke the `max` method using `ClassName.methodName` (i.e., `TestMax.max`).

5.3.1 Call Stacks

stack

Each time a method is invoked, the system stores parameters and variables in an area of memory known as a *stack*, which stores elements in last-in, first-out fashion. When a method calls another method, the caller's stack space is kept intact, and new space is created to handle the new method call. When a method finishes its work and returns to its caller, its associated space is released.

Understanding call stacks helps you to comprehend how methods are invoked. The variables defined in the `main` method are `i`, `j`, and `k`. The variables defined in the `max` method are `num1`, `num2`, and `result`. The variables `num1` and `num2` are defined in the method signature and are parameters of the method. Their values are passed through method invocation. Figure 5.3 illustrates the variables in the stack.



(a) The `main` method is invoked. (b) The `max` method is invoked. (c) The `max` method is being executed. (d) The `max` method is finished and the return value is sent to `k`. (e) The `main` method is finished.

FIGURE 5.3 When the `max` method is invoked, the flow of control transfers to the `max` method. Once the `max` method is finished, it returns control back to the caller.

5.4 `void` Method Example

Video Note
Use void method

The preceding section gives an example of a value-returning method. This section shows how to define and invoke a `void` method. Listing 5.2 gives a program that defines a method named `printGrade` and invokes it to print the grade for a given score.

LISTING 5.2 `TestVoidMethod.java`

`main` method

```
1 public class TestVoidMethod {
2   public static void main(String[] args) {
3     System.out.print("The grade is ");
```

```
4     printGrade(78.5);                                    invoke printGrade
5
6     System.out.print("The grade is ");
7     printGrade(59.5);                                    printGrade method
8   }
9
10  public static void printGrade(double score) {
11    if (score >= 90.0) {
12      System.out.println('A');
13    }
14    else if (score >= 80.0) {
15      System.out.println('B');
16    }
17    else if (score >= 70.0) {
18      System.out.println('C');
19    }
20    else if (score >= 60.0) {
21      System.out.println('D');
22    }
23    else {
24      System.out.println('F');
25    }
26  }
27 }
```

```
The grade is C
The grade is F
```



The **printGrade** method is a **void** method. It does not return any value. A call to a **void** method must be a statement. So, it is invoked as a statement in line 4 in the **main** method. Like any Java statement, it is terminated with a semicolon. invoke void method

To see the differences between a void and a value-returning method, let us redesign the **printGrade** method to return a value. The new method, which we call **getGrade**, returns the grade as shown in Listing 5.3. void vs. value-returned

LISTING 5.3 TestReturnGradeMethod.java

```
1 public class TestReturnGradeMethod {
2   public static void main(String[] args) {              main method
3     System.out.print("The grade is " + getGrade(78.5));
4     System.out.print("\nThe grade is " + getGrade(59.5));   invoke printGrade
5   }
6
7   public static char getGrade(double score) {           printGrade method
8     if (score >= 90.0)
9       return 'A';
10    else if (score >= 80.0)
11      return 'B';
12    else if (score >= 70.0)
13      return 'C';
14    else if (score >= 60.0)
15      return 'D';
16    else
17      return 'F';
18  }
19 }
```



```
The grade is C
The grade is F
```

The `getGrade` method defined in lines 7–18 returns a character grade based on the numeric score value. The caller invokes this method in lines 3–4.

The `getGrade` method can be invoked by a caller wherever a character may appear. The `printGrade` method does not return any value. It must be also invoked as a statement.



Note

return in void method

A `return` statement is not needed for a `void` method, but it can be used for terminating the method and returning to the method's caller. The syntax is simply

```
return;
```

This is not often done, but sometimes it is useful for circumventing the normal flow of control in a `void` method. For example, the following code has a return statement to terminate the method when the score is invalid.

```
public static void printGrade(double score) {
  if (score < 0 || score > 100) {
    System.out.println("Invalid score");
    return;
  }

  if (score >= 90.0) {
    System.out.println('A');
  }
  else if (score >= 80.0) {
    System.out.println('B');
  }
  else if (score >= 70.0) {
    System.out.println('C');
  }
  else if (score >= 60.0) {
    System.out.println('D');
  }
  else {
    System.out.println('F');
  }
}
```

5.5 Passing Parameters by Values

The power of a method is its ability to work with parameters. You can use `println` to print any string and `max` to find the maximum between any two `int` values. When calling a method, you need to provide arguments, which must be given in the same order as their respective parameters in the method signature. This is known as *parameter order association*. For example, the following method prints a message `n` times:

parameter order association

```
public static void nPrintln(String message, int n) {
  for (int i = 0; i < n; i++)
    System.out.println(message);
}
```

You can use `nPrintln("Hello", 3)` to print `"Hello"` three times. The `nPrintln("Hello", 3)` statement passes the actual string parameter `"Hello"` to the parameter `message`; passes `3` to `n`;

and prints `"Hello"` three times. However, the statement `nPrintln(3, "Hello")` would be wrong. The data type of `3` does not match the data type for the first parameter, `message`, nor does the second parameter, `"Hello"`, match the second parameter, `n`.



Caution

The arguments must match the parameters in *order*, *number*, and *compatible type*, as defined in the method signature. Compatible type means that you can pass an argument to a parameter without explicit casting, such as passing an `int` value argument to a `double` value parameter.

When you invoke a method with a parameter, the value of the argument is passed to the parameter. This is referred to as *pass-by-value*. If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter. The variable is not affected, regardless of the changes made to the parameter inside the method. As shown in Listing 5.4, the value of `x` (`1`) is passed to the parameter `n` to invoke the `increment` method (line 5). `n` is incremented by `1` in the method (line 10), but `x` is not changed no matter what the method does.

pass-by-value

LISTING 5.4 Increment.java

```
1 public class Increment {
2   public static void main(String[] args) {
3     int x = 1;
4     System.out.println("Before the call, x is " + x);
5     increment(x);
6     System.out.println("after the call, x is " + x);
7   }
8
9   public static void increment(int n) {
10    n++;
11    System.out.println("n inside the method is " + n);
12  }
13 }
```

invoke increment

increment **n**

```
Before the call, x is 1
n inside the method is 2
after the call, x is 1
```



Listing 5.5 gives another program that demonstrates the effect of passing by value. The program creates a method for swapping two variables. The `swap` method is invoked by passing two arguments. Interestingly, the values of the arguments are not changed after the method is invoked.

LISTING 5.5 TestPassByValue.java

```
1 public class TestPassByValue {
2   /** Main method */
3   public static void main(String[] args) {
4     // Declare and initialize variables
5     int num1 = 1;
6     int num2 = 2;
7
8     System.out.println("Before invoking the swap method, num1 is " +
9       num1 + " and num2 is " + num2);
10
11    // Invoke the swap method to attempt to swap two variables
12    swap(num1, num2);
```

false swap

```
13
14      System.out.println("After invoking the swap method, num1 is " +
15        num1 + " and num2 is " + num2);
16    }
17
18    /** Swap two variables */
19    public static void swap(int n1, int n2) {
20      System.out.println("\tInside the swap method");
21      System.out.println("\t\tBefore swapping n1 is " + n1
22        + " n2 is " + n2);
23
24      // Swap n1 with n2
25      int temp = n1;
26      n1 = n2;
27      n2 = temp;
28
29      System.out.println("\t\tAfter swapping n1 is " + n1
30        + " n2 is " + n2);
31    }
32  }
```



```
Before invoking the swap method, num1 is 1 and num2 is 2
  Inside the swap method
    Before swapping n1 is 1 n2 is 2
    After swapping n1 is 2 n2 is 1
After invoking the swap method, num1 is 1 and num2 is 2
```

Before the `swap` method is invoked (line 12), `num1` is `1` and `num2` is `2`. After the `swap` method is invoked, `num1` is still `1` and `num2` is still `2`. Their values have not been swapped. As shown in Figure 5.4, the values of the arguments `num1` and `num2` are passed to `n1` and `n2`, but `n1` and `n2` have their own memory locations independent of `num1` and `num2`. Therefore, changes in `n1` and `n2` do not affect the contents of `num1` and `num2`.

Another twist is to change the parameter name `n1` in `swap` to `num1`. What effect does this have? No change occurs, because it makes no difference whether the parameter and the argument have the same name. The parameter is a variable in the method with its own memory space. The variable is allocated when the method is invoked, and it disappears when the method is returned to its caller.

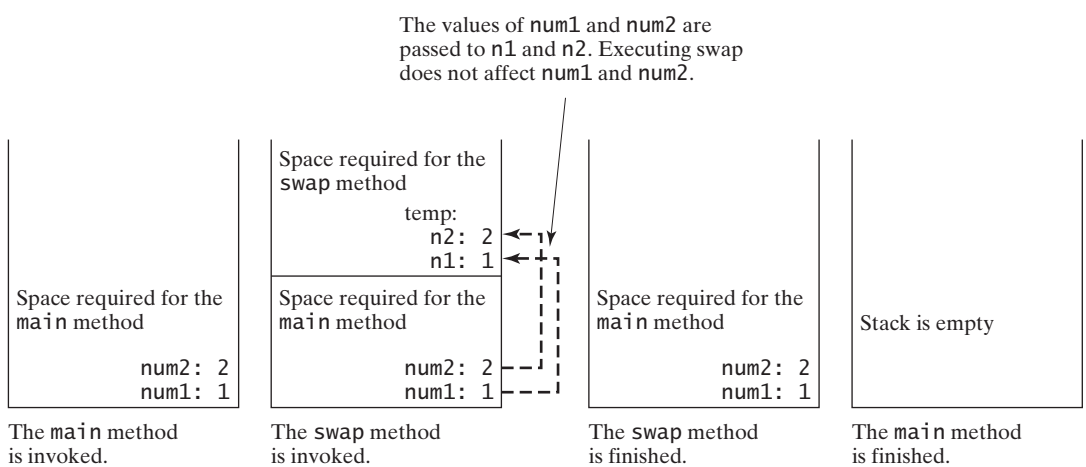

The values of num1 and num2 are passed to n1 and n2. Executing swap does not affect num1 and num2.
Space required for the main method
num2: 2
num1: 1
Space required for the swap method
temp:
n2: 2
n1: 1
Space required for the main method
num2: 2
num1: 1
Space required for the main method
num2: 2
num1: 1
Stack is empty
The main method is invoked.
The swap method is invoked.
The swap method is finished.
The main method is finished.

FIGURE 5.4 The values of the variables are passed to the parameters of the method.



Note

For simplicity, Java programmers often say *passing an argument* ***x*** *to a parameter* ***y***, which actually means *passing the value of* ***x*** *to* ***y***.

5.6 Modularizing Code



Video Note
Modularize code

Methods can be used to reduce redundant code and enable code reuse. Methods can also be used to modularize code and improve the quality of the program.

Listing 4.8 gives a program that prompts the user to enter two integers and displays their greatest common divisor. You can rewrite the program using a method, as shown in Listing 5.6.

LISTING 5.6 GreatestCommonDivisorMethod.java

```
1  import java.util.Scanner;
2
3  public class GreatestCommonDivisorMethod {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8
9      // Prompt the user to enter two integers
10     System.out.print("Enter first integer: ");
11     int n1 = input.nextInt();
12     System.out.print("Enter second integer: ");
13     int n2 = input.nextInt();
14
15     System.out.println("The greatest common divisor for " + n1 +
16       " and " + n2 + " is " + gcd(n1, n2) );                    invoke gcd
17   }
18
19   /** Return the gcd of two integers */
20   public static int gcd(int n1, int n2) {                      compute gcd
21     int gcd = 1; // Initial gcd is 1
22     int k = 2;  // Possible gcd
23
24     while (k <= n1 && k <= n2) {
25       if (n1 % k == 0 && n2 % k == 0)
26         gcd = k; // Update gcd
27       k++;
28     }
29
30     return gcd; // Return gcd                                  return gcd
31   }
32 }
```

```
Enter first integer: 45 Enter
Enter second integer: 75 Enter
The greatest common divisor for 45 and 75 is 15
```



By encapsulating the code for obtaining the gcd in a method, this program has several advantages:

1. It isolates the problem for computing the gcd from the rest of the code in the main method. Thus, the logic becomes clear and the program is easier to read.

2. The errors on computing gcd are confined in the `gcd` method, which narrows the scope of debugging.

3. The `gcd` method now can be reused by other programs.

Listing 5.7 applies the concept of code modularization to improve Listing 4.14, PrimeNumber.java.

LISTING 5.7 PrimeNumberMethod.java

```
1  public class PrimeNumberMethod {
2    public static void main(String[] args) {
3      System.out.println("The first 50 prime numbers are \n");
4      printPrimeNumbers(50);
5    }
6
7    public static void printPrimeNumbers(int numberOfPrimes) {
8      final int NUMBER_OF_PRIMES_PER_LINE = 10; // Display 10 per line
9      int count = 0; // Count the number of prime numbers
10     int number = 2; // A number to be tested for primeness
11
12     // Repeatedly find prime numbers
13     while (count < numberOfPrimes) {
14       // Print the prime number and increase the count
15       if (isPrime(number) ) {
16         count++; // Increase the count
17
18         if (count % NUMBER_OF_PRIMES_PER_LINE == 0) {
19           // Print the number and advance to the new line
20           System.out.printf("%-5s\n", number);
21         }
22         else
23           System.out.printf("%-5s", number);
24       }
25
26       // Check whether the next number is prime
27       number++;
28     }
29   }
30
31   /** Check whether number is prime */
32   public static boolean isPrime(int number) {
33     for (int divisor = 2; divisor <= number / 2; divisor++) {
34       if (number % divisor == 0) { // If true, number is not prime
35         return false; // number is not a prime
36       }
37     }
38
39     return true; // number is prime
40   }
41 }
```

invoke **printPrimeNumbers**

printPrimeNumbers method

invoke **isPrime**

isPrime method



```
The first 50 prime numbers are

2    3    5    7    11   13   17   19   23   29
31   37   41   43   47   53   59   61   67   71
73   79   83   89   97   101  103  107  109  113
127  131  137  139  149  151  157  163  167  173
179  181  191  193  197  199  211  223  227  229
```

We divided a large problem into two subproblems. As a result, the new program is easier to read and easier to debug. Moreover, the methods `printPrimeNumbers` and `isPrime` can be reused by other programs.

5.7 Problem: Converting Decimals to Hexadecimals

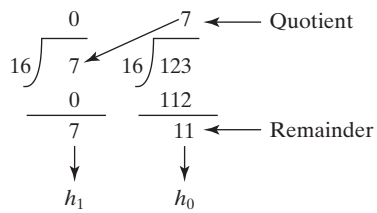
Hexadecimals are often used in computer systems programming. Appendix F introduces number systems. This section presents a program that converts a decimal to a hexadecimal.

To convert a decimal number d to a hexadecimal number is to find the hexadecimal digits $h_n, h_{n-1}, h_{n-2}, \ldots, h_2, h_1$, and h_0 such that

$$d = h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

These numbers can be found by successively dividing d by 16 until the quotient is 0. The remainders are $h_0, h_1, h_2, \ldots, h_{n-2}, h_{n-1}$, and h_n.

For example, the decimal number `123` is `7B` in hexadecimal. The conversion is done as follows:



Listing 5.8 gives a program that prompts the user to enter a decimal number and converts it into a hex number as a string.

LISTING 5.8 Decimal2HexConversion.java

```
1 import java.util.Scanner;
2
3 public class Decimal2HexConversion {
4   /** Main method */
5   public static void main(String[] args) {
6     // Create a Scanner
7     Scanner input = new Scanner(System.in);
8
9     // Prompt the user to enter a decimal integer
10    System.out.print("Enter a decimal number: ");
11    int decimal = input.nextInt();                          input string
12
13    System.out.println("The hex number for decimal " +
14      decimal + " is " + decimalToHex(decimal) );           hex to decimal
15  }
16
17  /** Convert a decimal to a hex as a string */
18  public static String decimalToHex(int decimal) {
19    String hex = "";
20
21    while (decimal != 0) {
22      int hexValue = decimal % 16;
23      hex = toHexChar(hexValue) + hex;
24      decimal = decimal / 16;
```

hex char to decimal
to uppercase

```
25     }
26
27     return hex;
28   }
29
30   /** Convert an integer to a single hex digit in a character */
31   public static char toHexChar(int hexValue) {
32     if (hexValue <= 9 && hexValue >= 0)
33       return (char)(hexValue + '0');
34     else  // hexValue <= 15 && hexValue >= 10
35       return (char)(hexValue - 10 + 'A');
36   }
37 }
```



```
Enter a decimal number: 1234 Enter
The hex number for decimal 1234 is 4D2
```



	line#	decimal	hex	hexValue	toHexChar(hexValue)
	19	1234	""		
iteration 1	22			2	
	23		"2"		2
	24	77			
iteration 2	22			13	
	23		"D2"		D
	24	4			
iteration 3	22			4	
	23		"4D2"		4
	24	0			

The program uses the `decimalToHex` method (lines 18–28) to convert a decimal integer to a hex number as a string. The method gets the remainder of the division of the decimal integer by `16` (line 22). The remainder is converted into a character by invoking the `toHexChar` method (line 23). The character is then appended to the hex string (line 23). The hex string is initially empty (line 19). Divide the decimal number by `16` to remove a hex digit from the number (line 24). The method repeatedly performs these operations in a loop until quotient becomes `0` (lines 21–25).

The `toHexChar` method (lines 31–36) converts a `hexValue` between `0` and `15` into a hex character. If `hexValue` is between `0` and `9`, it is converted to `(char)(hexValue + '0')` (line 33). Recall when adding a character with an integer, the character's Unicode is used in the evaluation. For example, if `hexValue` is `5`, `(char)(hexValue + '0')` returns `'5'`. Similarly, if `hexValue` is between `10` and `15`, it is converted to `(char)(hexValue + 'A')` (line 35). For example, if `hexValue` is `11`, `(char)(hexValue + 'A')` returns `'B'`.

5.8 Overloading Methods

The `max` method that was used earlier works only with the `int` data type. But what if you need to determine which of two floating-point numbers has the maximum value? The solution

is to create another method with the same name but different parameters, as shown in the following code:

```
public static double max(double num1, double num2) {
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

If you call `max` with `int` parameters, the `max` method that expects `int` parameters will be invoked; if you call `max` with `double` parameters, the `max` method that expects `double` parameters will be invoked. This is referred to as *method overloading*; that is, two methods have the same name but different parameter lists within one class. The Java compiler determines which method is used based on the method signature.

method overloading

Listing 5.9 is a program that creates three methods. The first finds the maximum integer, the second finds the maximum double, and the third finds the maximum among three double values. All three methods are named `max`.

LISTING 5.9 TestMethodOverloading.java

```
1  public class TestMethodOverloading {
2    /** Main method */
3    public static void main(String[] args) {
4      // Invoke the max method with int parameters
5      System.out.println("The maximum between 3 and 4 is "
6        + max(3, 4));
7
8      // Invoke the max method with the double parameters
9      System.out.println("The maximum between 3.0 and 5.4 is "
10       + max(3.0, 5.4));
11
12     // Invoke the max method with three double parameters
13     System.out.println("The maximum between 3.0, 5.4, and 10.14 is "
14       + max(3.0, 5.4, 10.14));
15   }
16
17   /** Return the max between two int values */
18   public static int max(int num1, int num2) {                    overloaded max
19     if (num1 > num2)
20       return num1;
21     else
22       return num2;
23   }
24
25   /** Find the max between two double values */
26   public static double max(double num1, double num2) {           overloaded max
27     if (num1 > num2)
28       return num1;
29     else
30       return num2;
31   }
32
33   /** Return the max among three double values */
34   public static double max(double num1, double num2, double num3) {   overloaded max
35     return max(max(num1, num2), num3);
36   }
37 }
```



```
The maximum between 3 and 4 is 4
The maximum between 3.0 and 5.4 is 5.4
The maximum between 3.0, 5.4, and 10.14 is 10.14
```

When calling `max(3, 4)` (line 6), the `max` method for finding the maximum of two integers is invoked. When calling `max(3.0, 5.4)` (line 10), the `max` method for finding the maximum of two doubles is invoked. When calling `max(3.0, 5.4, 10.14)` (line 14), the `max` method for finding the maximum of three double values is invoked.

Can you invoke the `max` method with an `int` value and a `double` value, such as `max(2, 2.5)`? If so, which of the `max` methods is invoked? The answer to the first question is yes. The answer to the second is that the `max` method for finding the maximum of two `double` values is invoked. The argument value 2 is automatically converted into a `double` value and passed to this method.

You may be wondering why the method `max(double, double)` is not invoked for the call `max(3, 4)`. Both `max(double, double)` and `max(int, int)` are possible matches for `max(3, 4)`. The Java compiler finds the most specific method for a method invocation. Since the method `max(int, int)` is more specific than `max(double, double)`, `max(int, int)` is used to invoke `max(3, 4)`.



Note

Overloaded methods must have different parameter lists. You cannot overload methods based on different modifiers or return types.



Note

ambiguous invocation

Sometimes there are two or more possible matches for an invocation of a method, but the compiler cannot determine the most specific match. This is referred to as *ambiguous invocation*. Ambiguous invocation causes a compile error. Consider the following code:

```
public class AmbiguousOverloading {
  public static void main(String[] args) {
    System.out.println(max(1, 2) );
  }

  public static double max(int num1, double num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }

  public static double max(double num1, int num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }
}
```

Both `max(int, double)` and `max(double, int)` are possible candidates to match `max(1, 2)`. Since neither is more specific than the other, the invocation is ambiguous, resulting in a compile error.

5.9 The Scope of Variables

The *scope of a variable* is the part of the program where the variable can be referenced. A variable defined inside a method is referred to as a *local variable*.

local variable

The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared and assigned a value before it can be used.

A parameter is actually a local variable. The scope of a method parameter covers the entire method.

A variable declared in the initial-action part of a `for`-loop header has its scope in the entire loop. But a variable declared inside a `for`-loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable, as shown in Figure 5.5.

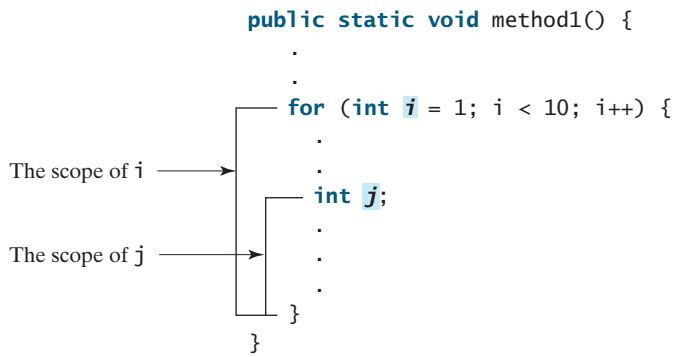


FIGURE 5.5 A variable declared in the initial action part of a `for`-loop header has its scope in the entire loop.

You can declare a local variable with the same name in different blocks in a method, but you cannot declare a local variable twice in the same block or in nested blocks, as shown in Figure 5.6.

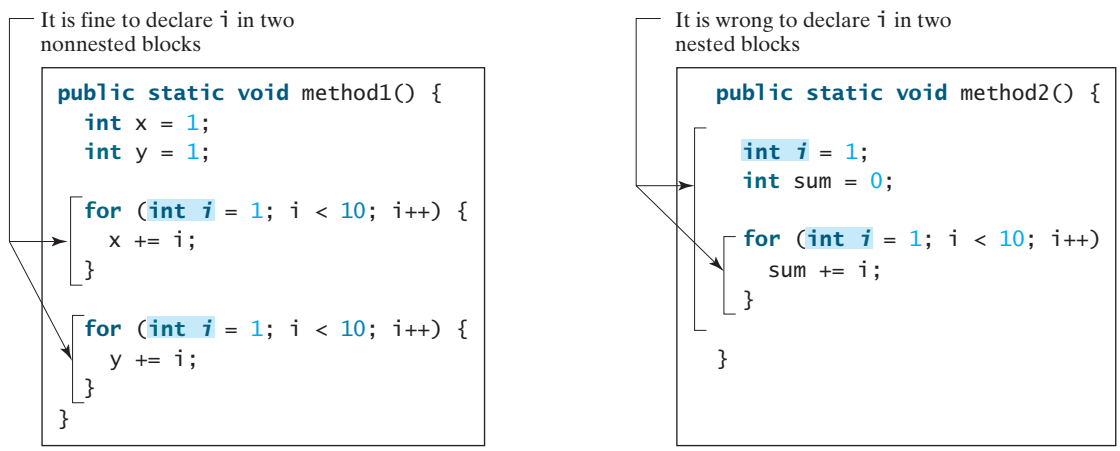

It is fine to declare i in two nonnested blocks
public static void method1() {
int x = 1;
int y = 1;
for (int i = 1; i < 10; i++) {
x += i;
}
for (int i = 1; i < 10; i++) {
y += i;
}
}
It is wrong to declare i in two nested blocks
public static void method2() {
int i = 1;
int sum = 0;
for (int i = 1; i < 10; i++)
sum += i;
}
}

FIGURE 5.6 A variable can be declared multiple times in nonnested blocks but only once in nested blocks.



Caution

Do not declare a variable inside a block and then attempt to use it outside the block. Here is an example of a common mistake:

```
for (int i = 0; i < 10; i++) {
}

System.out.println(i);
```

The last statement would cause a syntax error, because variable `i` is not defined outside of the `for` loop.

5.10 The `Math` Class

The `Math` class contains the methods needed to perform basic mathematical functions. You have already used the `pow(a, b)` method to compute a^b in Listing 2.8, ComputeLoan.java, and the `Math.random()` method in Listing 3.4, SubtractionQuiz.java. This section introduces other useful methods in the `Math` class. They can be categorized as *trigonometric methods*, *exponent methods*, and *service methods*. Besides methods, the `Math` class provides two useful `double` constants, `PI` and `E` (the base of natural logarithms). You can use these constants as `Math.PI` and `Math.E` in any program.

5.10.1 Trigonometric Methods

The `Math` class contains the following trigonometric methods:

```
/** Return the trigonometric sine of an angle in radians */
public static double sin(double radians)

/** Return the trigonometric cosine of an angle in radians */
public static double cos(double radians)

/** Return the trigonometric tangent of an angle in radians */
public static double tan(double radians)

/** Convert the angle in degrees to an angle in radians */
public static double toRadians(double degree)

/** Convert the angle in radians to an angle in degrees */
public static double toDegrees(double radians)

/** Return the angle in radians for the inverse of sin */
public static double asin(double a)

/** Return the angle in radians for the inverse of cos */
public static double acos(double a)

/** Return the angle in radians for the inverse of tan */
public static double atan(double a)
```

The parameter for `sin`, `cos`, and `tan` is an angle in radians. The return value for `asin`, `acos`, and `atan` is a degree in radians in the range between $-\pi/2$ and $\pi/2$. One degree is equal to $\pi/180$ in radians, 90 degrees is equal to $\pi/2$ in radians, and 30 degrees is equal to $\pi/6$ in radians.

For example,

```
Math.toDegrees(Math.PI / 2) returns 90.0
Math.toRadians(30) returns π/6
Math.sin(0) returns 0.0
Math.sin(Math.toRadians(270)) returns -1.0
Math.sin(Math.PI / 6) returns 0.5
Math.sin(Math.PI / 2) returns 1.0
Math.cos(0) returns 1.0
Math.cos(Math.PI / 6) returns 0.866
Math.cos(Math.PI / 2) returns 0
Math.asin(0.5) returns π/6
```

5.10.2 Exponent Methods

There are five methods related to exponents in the **Math** class:

```
/** Return e raised to the power of x (e^x) */
public static double exp(double x)

/** Return the natural logarithm of x (ln(x) = log_e(x)) */
public static double log(double x)

/** Return the base 10 logarithm of x (log_10(x)) */
public static double log10(double x)

/** Return a raised to the power of b (a^b) */
public static double pow(double a, double b)

/** Return the square root of x (√x) for x >= 0 */
public static double sqrt(double x)
```

For example,

```
Math.exp(1) returns 2.71828
Math.log(Math.E) returns 1.0
Math.log10(10) returns 1.0
Math.pow(2, 3) returns 8.0
Math.pow(3, 2) returns 9.0
Math.pow(3.5, 2.5) returns 22.91765
Math.sqrt(4) returns 2.0
Math.sqrt(10.5) returns 3.24
```

5.10.3 The Rounding Methods

The **Math** class contains five rounding methods:

```
/** x is rounded up to its nearest integer. This integer is
  * returned as a double value. */
public static double ceil(double x)

/** x is rounded down to its nearest integer. This integer is
  * returned as a double value. */
public static double floor(double x)

/** x is rounded to its nearest integer. If x is equally close
  * to two integers, the even one is returned as a double. */
public static double rint(double x)

/** Return (int)Math.floor(x + 0.5). */
public static int round(float x)

/** Return (long)Math.floor(x + 0.5). */
public static long round(double x)
```

For example,

```
Math.ceil(2.1) returns 3.0
Math.ceil(2.0) returns 2.0
Math.ceil(-2.0) returns -2.0
Math.ceil(-2.1) returns -2.0
Math.floor(2.1) returns 2.0
Math.floor(2.0) returns 2.0
```

```
Math.floor(-2.0) returns -2.0
Math.floor(-2.1) returns -3.0
Math.rint(2.1) returns 2.0
Math.rint(-2.0) returns -2.0
Math.rint(-2.1) returns -2.0
Math.rint(2.5) returns 2.0
Math.rint(3.5) returns 4.0
Math.rint(-2.5) returns -2.0
Math.round(2.6f) returns 3 // Returns int
Math.round(2.0) returns 2  // Returns long
Math.round(-2.0f) returns -2
Math.round(-2.6) returns -3
```

5.10.4 The `min`, `max`, and `abs` Methods

The `min` and `max` methods are overloaded to return the minimum and maximum numbers between two numbers (`int`, `long`, `float`, or `double`). For example, `max(3.4, 5.0)` returns `5.0`, and `min(3, 2)` returns `2`.

The `abs` method is overloaded to return the absolute value of the number (`int`, `long`, `float`, and `double`). For example,

```
Math.max(2, 3) returns 3
Math.max(2.5, 3) returns 3.0
Math.min(2.5, 3.6) returns 2.5
Math.abs(-2) returns 2
Math.abs(-2.1) returns 2.1
```

5.10.5 The `random` Method

You have used the `random()` method to generate a random `double` value greater than or equal to 0.0 and less than 1.0 (`0 <= Math.random() < 1.0`). This method is very useful. You can use it to write a simple expression to generate random numbers in any range. For example,

`(int) (Math.random( ) * 10)` ⟶ Returns a random integer between `0` and `9`

`50 + (int) (Math.random( ) * 50)` ⟶ Returns a random integer between `50` and `99`

In general,

`a + Math.random( ) * b` ⟶ Returns a random number between `a` and `a + b` excluding `a + b`

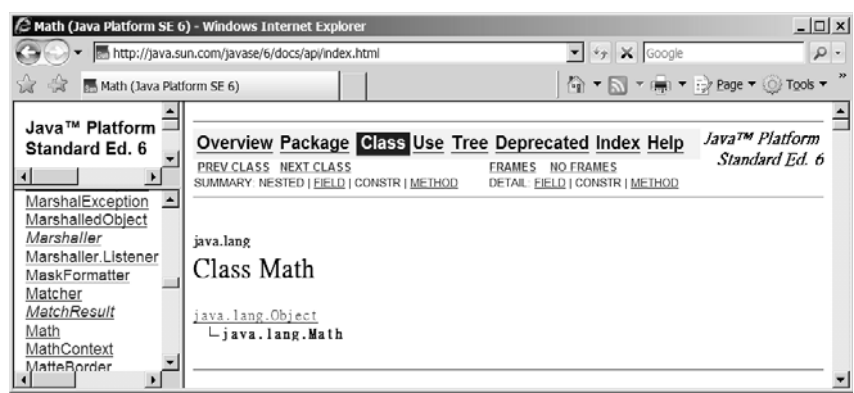


FIGURE 5.7 You can view the documentation for Java API online.



Tip

You can view the complete documentation for the `Math` class online at http://java.sun.com/javase/6/docs/api/index.html, as shown in Figure 5.7.



Note

Not all classes need a `main` method. The `Math` class and `JOptionPane` class do not have `main` methods. These classes contain methods for other classes to use.

5.11 Case Study: Generating Random Characters

Computer programs process numerical data and characters. You have seen many examples that involve numerical data. It is also important to understand characters and how to process them. This section presents an example for generating random characters.

As introduced in §2.13, every character has a unique Unicode between 0 and FFFF in hexadecimal (65535 in decimal). To generate a random character is to generate a random integer between 0 and 65535 using the following expression (note that since `0 <= Math.random() < 1.0`, you have to add 1 to 65535):

```
(int)(Math.random() * (65535 + 1))
```

Now let us consider how to generate a random lowercase letter. The Unicodes for lowercase letters are consecutive integers starting from the Unicode for `'a'`, then that for `'b'`, `'c'`, . . . , and `'z'`. The Unicode for `'a'` is

```
(int)'a'
```

So a random integer between `(int)'a'` and `(int)'z'` is

```
(int)((int)'a' + Math.random() * ((int)'z' - (int)'a' + 1)
```

As discussed in §2.13.3, all numeric operators can be applied to the `char` operands. The `char` operand is cast into a number if the other operand is a number or a character. Thus the preceding expression can be simplified as follows:

```
'a' + Math.random() * ('z' - 'a' + 1)
```

and a random lowercase letter is

```
(char)('a' + Math.random() * ('z' - 'a' + 1))
```

To generalize the foregoing discussion, a random character between any two characters `ch1` and `ch2` with `ch1 < ch2` can be generated as follows:

```
(char)(ch1 + Math.random() * (ch2 - ch1 + 1))
```

This is a simple but useful discovery. Let us create a class named `RandomCharacter` in Listing 5.10 with five overloaded methods to get a certain type of character randomly. You can use these methods in your future projects.

LISTING 5.10 RandomCharacter.java

```
1 public class RandomCharacter {
2   /** Generate a random character between ch1 and ch2 */
3   public static char getRandomCharacter(char ch1, char ch2) {          getRandomCharacter
4     return (char)(ch1 + Math.random() * (ch2 - ch1 + 1));
5   }
```

getRandomLowerCaseLetter()

getRandomUpperCaseLetter()

getRandomDigitCharacter()

getRandomCharacter()

```
 6
 7   /** Generate a random lowercase letter */
 8   public static char getRandomLowerCaseLetter() {
 9     return getRandomCharacter('a', 'z');
10   }
11
12   /** Generate a random uppercase letter */
13   public static char getRandomUpperCaseLetter() {
14     return getRandomCharacter('A', 'Z');
15   }
16
17   /** Generate a random digit character */
18   public static char getRandomDigitCharacter() {
19     return getRandomCharacter('0', '9');
20   }
21
22   /** Generate a random character */
23   public static char getRandomCharacter() {
24     return getRandomCharacter('\u0000', '\uFFFF');
25   }
26 }
```

Listing 5.11 gives a test program that displays 175 random lowercase letters.

LISTING 5.11 TestRandomCharacter.java

constants

lower-case letter

```
1 public class TestRandomCharacter {
2   /** Main method */
3   public static void main(String[] args) {
4     final int NUMBER_OF_CHARS = 175;
5     final int CHARS_PER_LINE = 25;
6
7     // Print random characters between 'a' and 'z', 25 chars per line
8     for (int i = 0; i < NUMBER_OF_CHARS; i++) {
9       char ch = RandomCharacter.getRandomLowerCaseLetter();
10      if ((i + 1) % CHARS_PER_LINE == 0)
11        System.out.println(ch);
12      else
13        System.out.print(ch);
14    }
15  }
16 }
```



```
gmjsohezfkgtazqgmswfclrao
pnrunulnwmaztlfjedmpchcif
lalqdgivxkxpbzulrmqmbhikr
lbnrjlsopfxahssqhwuuljvbe
xbhdotzhpehbqmuwsfktwsoli
cbuwkzgxpmtzihgatdslvbwbz
bfesoklwbhnooygiigzdxuqni
```

Line 9 invokes `getRandomLowerCaseLetter()` defined in the `RandomCharacter` class. Note that `getRandomLowerCaseLetter()` does not have any parameters, but you still have to use the parentheses when defining and invoking the method.

parentheses required

5.12 Method Abstraction and Stepwise Refinement

method abstraction

The key to developing software is to apply the concept of abstraction. You will learn many levels of abstraction from this book. *Method abstraction* is achieved by separating the use of a

method from its implementation. The client can use a method without knowing how it is implemented. The details of the implementation are encapsulated in the method and hidden from the client who invokes the method. This is known as *information hiding* or *encapsulation*. If you decide to change the implementation, the client program will not be affected, provided that you do not change the method signature. The implementation of the method is hidden from the client in a "black box," as shown in Figure 5.8.

information hiding

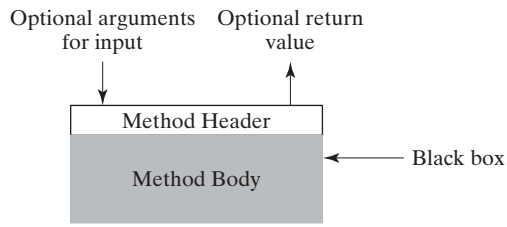


FIGURE 5.8 The method body can be thought of as a black box that contains the detailed implementation for the method.

You have already used the `System.out.print` method to display a string, the `JOptionPane.showInputDialog` method to read a string from a dialog box, and the `max` method to find the maximum number. You know how to write the code to invoke these methods in your program, but as a user of these methods, you are not required to know how they are implemented.

The concept of method abstraction can be applied to the process of developing programs. When writing a large program, you can use the *divide-and-conquer* strategy, also known as *stepwise refinement*, to decompose it into subproblems. The subproblems can be further decomposed into smaller, more manageable problems.

divide and conquer
stepwise refinement

Suppose you write a program that displays the calendar for a given month of the year. The program prompts the user to enter the year and the month, then displays the entire calendar for the month, as shown in the following sample run:



```
Enter full year (e.g., 2001): 2006 Enter
Enter month in number between 1 and 12: 6 Enter
           June 2006
---------------------------------
 Sun  Mon  Tue  Wed  Thu  Fri  Sat
                     1    2    3
   4    5    6    7    8    9   10
  11   12   13   14   15   16   17
  18   19   20   21   22   23   24
  25   26   27   28   29   30
```

Let us use this example to demonstrate the divide-and-conquer approach.

5.12.1 Top-Down Design

How would you get started on such a program? Would you immediately start coding? Beginning programmers often start by trying to work out the solution to every detail. Although details are important in the final program, concern for detail in the early stages may block the problem-solving process. To make problem solving flow as smoothly as possible, this example begins by using method abstraction to isolate details from design and only later implements the details.

For this example, the problem is first broken into two subproblems: get input from the user, and print the calendar for the month. At this stage, you should be concerned with what the subproblems will achieve, not with how to get input and print the calendar for the month. You can draw a structure chart to help visualize the decomposition of the problem (see Figure 5.9(a)).

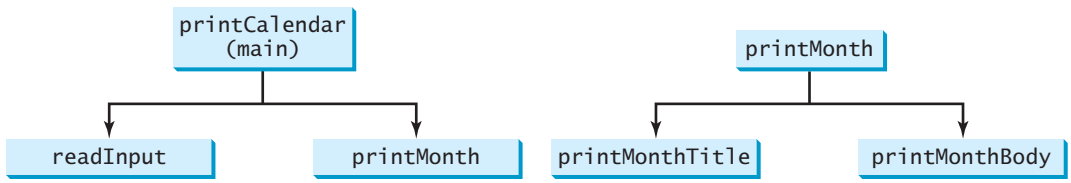


FIGURE 5.9 The structure chart shows that the `printCalendar` problem is divided into two subproblems, `readInput` and `printMonth`, and that `printMonth` is divided into two smaller subproblems, `printMonthTitle` and `printMonthBody`.

The problem of printing the calendar for a given month can be broken into two subproblems: print the month title, and print the month body, as shown in Figure 5.9(b). The month title consists of three lines: month and year, a dash line, and the names of the seven days of the week. You need to get the month name (e.g., January) from the numeric month (e.g., 1). This is accomplished in `getMonthName` (see Figure 5.10(a)).

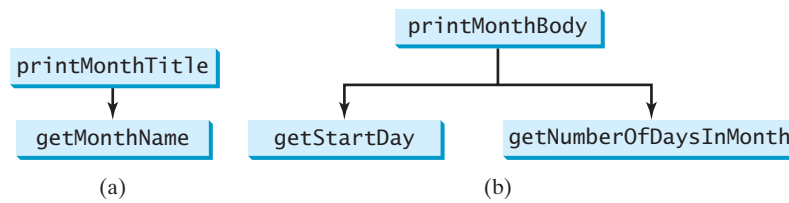


FIGURE 5.10 (a) To `printMonthTitle`, you need `getMonthName`. (b) The `printMonthBody` problem is refined into several smaller problems.

In order to print the month body, you need to know which day of the week is the first day of the month (`getStartDay`) and how many days the month has (`getNumberOfDaysInMonth`), as shown in Figure 5.10(b). For example, December 2005 has 31 days, and December 1, 2005, is Thursday.

How would you get the start day for the first date in a month? There are several ways to do so. For now, we'll use the following approach. Assume you know that the start day (`startDay1800 = 3`) for Jan 1, 1800, was Wednesday. You could compute the total number of

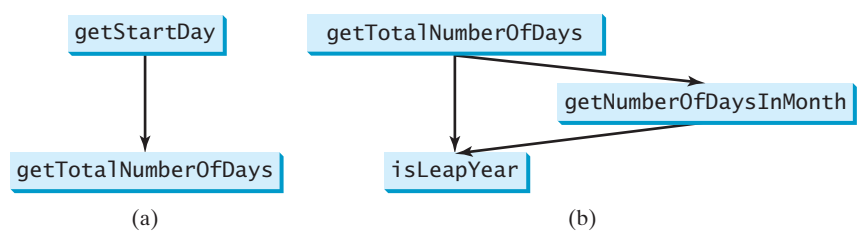


FIGURE 5.11 (a) To `getStartDay`, you need `getTotalNumberOfDays`. (b) The `getTotalNumberOfDays` problem is refined into two smaller problems.

days (`totalNumberOfDays`) between Jan 1, 1800, and the first date of the calendar month. The start day for the calendar month is `(totalNumberOfDays + startDay1800) % 7`, since every week has seven days. So the `getStartDay` problem can be further refined as `getTotalNumberOfDays`, as shown in Figure 5.11(a).

To get the total number of days, you need to know whether the year is a leap year and the number of days in each month. So `getTotalNumberOfDays` is further refined into two subproblems: `isLeapYear` and `getNumberOfDaysInMonth`, as shown in Figure 5.11(b). The complete structure chart is shown in Figure 5.12.

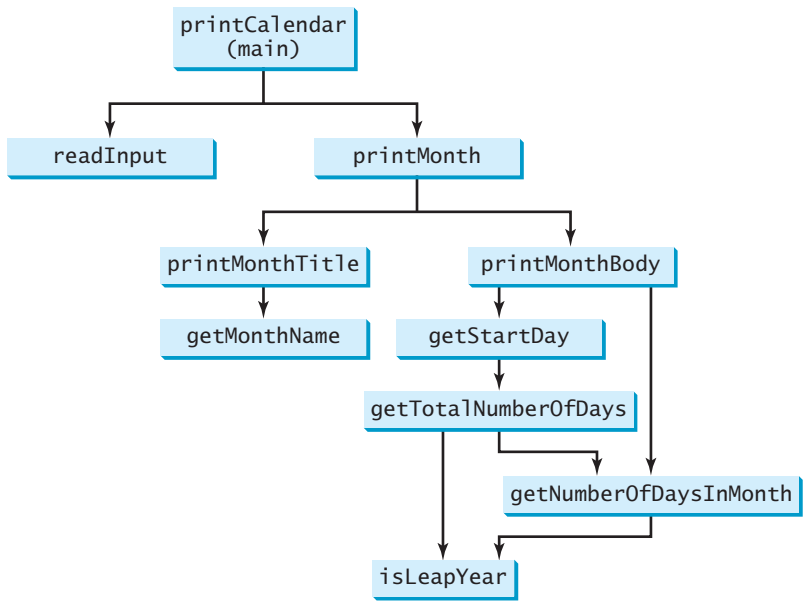


FIGURE 5.12 The structure chart shows the hierarchical relationship of the subproblems in the program.

5.12.2 Top-Down or Bottom-Up Implementation

Now we turn our attention to implementation. In general, a subproblem corresponds to a method in the implementation, although some are so simple that this is unnecessary. You would need to decide which modules to implement as methods and which to combine in other methods. Decisions of this kind should be based on whether the overall program will be easier to read as a result of your choice. In this example, the subproblem `readInput` can be simply implemented in the `main` method.

You can use either a "top-down" or a "bottom-up" approach. The top-down approach implements one method in the structure chart at a time from the top to the bottom. Stubs can be used for the methods waiting to be implemented. A *stub* is a simple but incomplete version of a method. The use of stubs enables you to quickly build the framework of the program. Implement the `main` method first, then use a stub for the `printMonth` method. For example, let `printMonth` display the year and the month in the stub. Thus, your program may begin like this:

top-down approach

stub

```
public class PrintCalendar {
  /** Main method */
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter year
    System.out.print("Enter full year (e.g., 2001): ");
    int year = input.nextInt();
```

```
    // Prompt the user to enter month
    System.out.print("Enter month as number between 1 and 12: ");
    int month = input.nextInt();

    // Print calendar for the month of the year
    printMonth(year, month);
  }

  /** A stub for printMonth may look like this */
  public static void printMonth(int year, int month) {
    System.out.print(month + " " + year);
  }

  /** A stub for printMonthTitle may look like this */
  public static void printMonthTitle(int year, int month) {
  }

  /** A stub for getMonthBody may look like this */
  public static void printMonthBody(int year, int month) {
  }

  /** A stub for getMonthName may look like this */
  public static String getMonthName(int month) {
    return "January"; // A dummy value
  }

  /** A stub for getMonthName may look like this */
  public static int getStartDay(int year, int month) {
    return 1; // A dummy value
  }

  /** A stub for getTotalNumberOfDays may look like this */
  public static int getTotalNumberOfDays(int year, int month) {
    return 10000; // A dummy value
  }

  /** A stub for getNumberOfDaysInMonth may look like this */
  public static int getNumberOfDaysInMonth(int year, int month) {
    return 31; // A dummy value
  }

  /** A stub for getTotalNumberOfDays may look like this */
  public static boolean isLeapYear(int year) {
    return true; // A dummy value
  }
}
```

Compile and test the program, and fix any errors. You can now implement the `printMonth` method. For methods invoked from the `printMonth` method, you can again use stubs.

bottom-up approach

The bottom-up approach implements one method in the structure chart at a time from the bottom to the top. For each method implemented, write a test program to test it. The top-down and bottom-up approaches are both fine. Both approaches implement methods incrementally, help to isolate programming errors, and make debugging easy. Sometimes they can be used together.

5.12.3 Implementation Details

The `isLeapYear(int year)` method can be implemented using the following code:

```
return (year % 400 == 0 || (year % 4 == 0 && year % 100 != 0));
```

Use the following facts to implement `getTotalNumberOfDaysInMonth(int year, int month)`:

- January, March, May, July, August, October, and December have 31 days.
- April, June, September, and November have 30 days.
- February has 28 days during a regular year and 29 days during a leap year. A regular year, therefore, has 365 days, a leap year 366 days.

To implement `getTotalNumberOfDays(int year, int month)`, you need to compute the total number of days (`totalNumberOfDays`) between January 1, 1800, and the first day of the calendar month. You could find the total number of days between the year 1800 and the calendar year and then figure out the total number of days prior to the calendar month in the calendar year. The sum of these two totals is `totalNumberOfDays`.

To print a body, first pad some space before the start day and then print the lines for every week.

The complete program is given in Listing 5.12.

LISTING 5.12 PrintCalendar.java

```
 1 import java.util.Scanner;
 2
 3 public class PrintCalendar {
 4   /** Main method */
 5   public static void main(String[] args) {
 6     Scanner input = new Scanner(System.in);
 7
 8     // Prompt the user to enter year
 9     System.out.print("Enter full year (e.g., 2001): ");
10     int year = input.nextInt();
11
12     // Prompt the user to enter month
13     System.out.print("Enter month in number between 1 and 12: ");
14     int month = input.nextInt();
15
16     // Print calendar for the month of the year
17     printMonth(year, month);
18   }
19
20   /** Print the calendar for a month in a year */
21   public static void printMonth(int year, int month) {                printMonth
22     // Print the headings of the calendar
23     printMonthTitle(year, month);
24
25     // Print the body of the calendar
26     printMonthBody(year, month);
27   }
28
29   /** Print the month title, e.g., May, 1999 */
30   public static void printMonthTitle(int year, int month) {           printMonthTitle
31     System.out.println("         " + getMonthName(month)
32       + " " + year);
33     System.out.println("-----------------------------");
34     System.out.println(" Sun Mon Tue Wed Thu Fri Sat");
35   }
36
37   /** Get the English name for the month */
38   public static String getMonthName(int month) {                      getMonthName
39     String monthName = " ";
```

```
40      switch (month) {
41        case 1: monthName = "January"; break;
42        case 2: monthName = "February"; break;
43        case 3: monthName = "March"; break;
44        case 4: monthName = "April"; break;
45        case 5: monthName = "May"; break;
46        case 6: monthName = "June"; break;
47        case 7: monthName = "July"; break;
48        case 8: monthName = "August"; break;
49        case 9: monthName = "September"; break;
50        case 10: monthName = "October"; break;
51        case 11: monthName = "November"; break;
52        case 12: monthName = "December";
53      }
54
55      return monthName;
56    }
57
58    /** Print month body */
59    public static void printMonthBody(int year, int month) {
60      // Get start day of the week for the first date in the month
61      int startDay = getStartDay(year, month);
62
63      // Get number of days in the month
64      int numberOfDaysInMonth = getNumberOfDaysInMonth(year, month);
65
66      // Pad space before the first day of the month
67      int i = 0;
68      for (i = 0; i < startDay; i++)
69        System.out.print("    ");
70
71      for (i = 1; i <= numberOfDaysInMonth; i++) {
72        System.out.printf("%4d", i);
73
74        if ((i + startDay) % 7 == 0)
75          System.out.println();
76      }
77
78      System.out.println();
79    }
80
81    /** Get the start day of month/1/year */
82    public static int getStartDay(int year, int month) {
83      final int START_DAY_FOR_JAN_1_1800 = 3;
84      // Get total number of days from 1/1/1800 to month/1/year
85      int totalNumberOfDays = getTotalNumberOfDays(year, month);
86
87      // Return the start day for month/1/year
88      return (totalNumberOfDays + START_DAY_FOR_JAN_1_1800) % 7;
89    }
90
91    /** Get the total number of days since January 1, 1800 */
92    public static int getTotalNumberOfDays(int year, int month) {
93      int total = 0;
94
95      // Get the total days from 1800 to 1/1/year
96      for (int i = 1800; i < year; i++)
97        if (isLeapYear(i))
```

getMonthBody

getStartDay

getTotalNumberOfDays

```
98          total = total + 366;
99        else
100         total = total + 365;
101
102     // Add days from Jan to the month prior to the calendar month
103     for (int i = 1; i < month; i++)
104       total = total + getNumberOfDaysInMonth(year, i);
105
106     return total;
107   }
108
109   /** Get the number of days in a month */
110   public static int getNumberOfDaysInMonth(int year, int month) {      getNumberOfDaysInMonth
111     if (month == 1 || month == 3 || month == 5 || month == 7 ||
112      month == 8 || month == 10 || month == 12)
113      return 31;
114
115     if (month == 4 || month == 6 || month == 9 || month == 11)
116       return 30;
117
118     if (month == 2) return isLeapYear(year) ? 29 : 28;
119
120     return 0; // If month is incorrect
121   }
122
123   /** Determine if it is a leap year */
124   public static boolean isLeapYear(int year) {                          isLeapYear
125     return year % 400 == 0 || (year % 4 == 0 && year % 100 != 0);
126   }
127 }
```

The program does not validate user input. For instance, if the user enters either a month not in the range between 1 and 12 or a year before 1800, the program displays an erroneous calendar. To avoid this error, add an `if` statement to check the input before printing the calendar.

This program prints calendars for a month but could easily be modified to print calendars for a whole year. Although it can print months only after January 1800, it could be modified to trace the day of a month before 1800.



Note

Method abstraction modularizes programs in a neat, hierarchical manner. Programs written as collections of concise methods are easier to write, debug, maintain, and modify. This writing style also promotes method reusability.



Tip

incremental development and testing

When implementing a large program, use the top-down or bottom-up approach. Do not write the entire program at once. Using these approaches seems to take more development time (because you repeatedly compile and run the program), but it actually saves time and makes debugging easier.

Key Terms

actual parameter 157
argument 157
ambiguous invocation 170
divide and conquer 177
formal parameter (i.e., parameter) 157
information hiding 177
method 156
method abstraction 176

method overloading 169
method signature 157
modifier 157
pass-by-value 163
parameter 157
return type 170
return value 157
scope of variable 171
stepwise refinement 177
stub 179

Chapter Summary

1. Making programs modular and reusable is one of the central goals in software engineering. Java provides many powerful constructs that help to achieve this goal. Methods are one such construct.

2. The method header specifies the *modifiers*, *return value type*, *method name*, and *parameters* of the method. The static modifier is used for all the methods in this chapter.

3. A method may return a value. The `returnValueType` is the data type of the value the method returns. If the method does not return a value, the `returnValueType` is the keyword `void`.

4. The *parameter list* refers to the type, order, and number of the parameters of a method. The method name and the parameter list together constitute the *method signature*. Parameters are optional; that is, a method may contain no parameters.

5. A return statement can also be used in a `void` method for terminating the method and returning to the method's caller. This is useful occasionally for circumventing the normal flow of control in a method.

6. The arguments that are passed to a method should have the same number, type, and order as the parameters in the method signature.

7. When a program calls a method, program control is transferred to the called method. A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached.

8. A value-returning method can also be invoked as a statement in Java. In this case, the caller simply ignores the return value.

9. Each time a method is invoked, the system stores parameters and local variables in a space known as a *stack*. When a method calls another method, the caller's stack space is kept intact, and new space is created to handle the new method call. When a method finishes its work and returns to its caller, its associated space is released.

10. A method can be overloaded. This means that two methods can have the same name, as long as their method parameter lists differ.

11. A variable declared in a method is called a local variable. The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared and initialized before it is used.

12. *Method abstraction* is achieved by separating the use of a method from its implementation. The client can use a method without knowing how it is implemented. The details of the implementation are encapsulated in the method and hidden from the client who invokes the method. This is known as *information hiding* or *encapsulation*.

13. Method abstraction modularizes programs in a neat, hierarchical manner. Programs written as collections of concise methods are easier to write, debug, maintain, and modify than would otherwise be the case. This writing style also promotes method reusability.

14. When implementing a large program, use the top-down or bottom-up coding approach. Do not write the entire program at once. This approach seems to take more time for coding (because you are repeatedly compiling and running the program), but it actually saves time and makes debugging easier.

Review Questions

Sections 5.2–5.4

5.1 What are the benefits of using a method? How do you define a method? How do you invoke a method?

5.2 What is the `return` type of a `main` method?

5.3 Can you simplify the `max` method in Listing 5.1 using the conditional operator?

5.4 True or false? A call to a method with a `void` return type is always a statement itself, but a call to a value-returning method is always a component of an expression.

5.5 What would be wrong with not writing a `return` statement in a value-returning method? Can you have a `return` statement in a `void` method? Does the `return` statement in the following method cause syntax errors?

```
public static void xMethod(double x, double y) {
  System.out.println(x + y);
  return x + y;
}
```

5.6 Define the terms parameter, argument, and method signature.

5.7 Write method headers for the following methods:

- Computing a sales commission, given the sales amount and the commission rate.
- Printing the calendar for a month, given the month and year.
- Computing a square root.
- Testing whether a number is even, and returning `true` if it is.
- Printing a message a specified number of times.
- Computing the monthly payment, given the loan amount, number of years, and annual interest rate.
- Finding the corresponding uppercase letter, given a lowercase letter.

5.8 Identify and correct the errors in the following program:

```
1 public class Test {
2   public static method1(int n, m) {
3     n += m;
4     method2(3.4);
5   }
6
7   public static int method2(int n) {
8     if (n > 0) return 1;
9     else if (n == 0) return 0;
10    else if (n < 0) return -1;
11  }
12 }
```

5.9 Reformat the following program according to the programming st yle and documentation guidelines proposed in §2.16, "Programming Style and Documentation." Use the next-line brace style.

```
public class Test {
  public static double method1(double i,double j)
  {
  while (i<j) {
    j--;
  }

  return j;
  }
}
```

Sections 5.5–5.7

5.10 How is an argument passed to a method? Can the argument have the same name as its parameter?

5.11 What is pass-by-value? Show the result of the following programs:

```
public class Test {
  public static void main(String[] args) {
    int max = 0;
    max(1, 2, max);
    System.out.println(max);
  }

  public static void max(
      int value1, int value2, int max) {
    if (value1 > value2)
      max = value1;
    else
      max = value2;
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    int i = 1;
    while (i <= 6) {
      method1(i, 2);
      i++;
    }
  }

  public static void method1(
      int i, int num) {
    for (int j = 1; j <= i; j++) {
      System.out.print(num + " ");
      num *= 2;
    }

    System.out.println();
  }
}
```

(b)

```
public class Test {
  public static void main(String[] args) {
    // Initialize times
    int times = 3;
    System.out.println("Before the call,"
      + " variable times is " + times);

    // Invoke nPrintln and display times
    nPrintln("Welcome to Java!", times);
    System.out.println("After the call,"
      + " variable times is " + times);
  }

  // Print the message n times
  public static void nPrintln(
      String message, int n) {
    while (n > 0) {
      System.out.println("n = " + n);
      System.out.println(message);
      n--;
    }
  }
}
```

(c)

```
public class Test {
  public static void main(String[] args) {
    int i = 0;
    while (i <= 4) {
      method1(i);
      i++;
    }

    System.out.println("i is " + i);
  }

  public static void method1(int i) {
    do {
      if (i % 3 != 0)
        System.out.print(i + " ");
      i--;
    }
    while (i >= 1);

    System.out.println();
  }
}
```

(d)

5.12 For (a) in the preceding question, show the contents of the stack just before the method `max` is invoked, just as `max` is entered, just before `max` is returned, and right after `max` is returned.

Section 5.8

5.13 What is method overloading? Is it permissible to define two methods that have the same name but different parameter types? Is it permissible to define two methods in a class that have identical method names and parameter lists but different return value types or different modifiers?

5.14 What is wrong in the following program?

```
public class Test {
  public static void method(int x) {
  }

  public static int method(int y) {
    return y;
  }
}
```

Section 5.9

5.15 Identify and correct the errors in the following program:

```
1 public class Test {
2   public static void main(String[] args) {
3     nPrintln("Welcome to Java!", 5);
4   }
5
6   public static void nPrintln(String message, int n) {
7     int n = 1;
```

```
 8     for (int i = 0; i < n; i++)
 9       System.out.println(message);
10   }
11 }
```

Section 5.10

5.16 True or false? The argument for trigonometric methods represents an angle in radians.

5.17 Write an expression that returns a random integer between 34 and 55. Write an expression that returns a random integer between 0 and 999. Write an expression that returns a random number between 5.5 and 55.5. Write an expression that returns a random lowercase letter.

5.18 Evaluate the following method calls:

(a) Math.sqrt(4)
(b) Math.sin(2 * Math.PI)
(c) Math.cos(2 * Math.PI)
(d) Math.pow(2, 2)
(e) Math.log(Math.E)
(f) Math.exp(1)
(g) Math.max(2, Math.min(3, 4))
(h) Math.rint(−2.5)
(i) Math.ceil(−2.5)
(j) Math.floor(−2.5)
(k) Math.round(−2.5F)
(l) Math.round(−2.5)
(m) Math.rint(2.5)
(n) Math.ceil(2.5)
(o) Math.floor(2.5)
(p) Math.round(2.5F)
(q) Math.round(2.5)
(r) Math.round(Math.abs(−2.5))

Programming Exercises

Sections 5.2–5.9

5.1 (*Math: pentagonal numbers*) A pentagonal number is defined as $n(3n-1)/2$ for $n = 1, 2, \ldots$, and so on. So, the first few numbers are 1, 5, 12, 22, ... Write the following method that returns a pentagonal number:

```
public static int getPentagonalNumber(int n)
```

Write a test program that displays the first 100 pentagonal numbers with 10 numbers on each line.

5.2* (*Summing the digits in an integer*) Write a method that computes the sum of the digits in an integer. Use the following method header:

```
public static int sumDigits(long n)
```

For example, `sumDigits(234)` returns 9 (2 + 3 + 4).

(*Hint*: Use the `%` operator to extract digits, and the `/` operator to remove the extracted digit. For instance, to extract 4 from 234, use `234 % 10` (= 4). To remove 4 from 234, use `234 / 10` (= 23). Use a loop to repeatedly extract and remove the digit until all the digits are extracted. Write a test program that prompts the user to enter an integer and displays the sum of all its digits.)

5.3** (*Palindrome integer*) Write the following two methods

```
// Return the reversal of an integer, i.e. reverse(456) returns 654
public static int reverse(int number)

// Return true if number is palindrome
public static boolean isPalindrome(int number)
```

Use the `reverse` method to implement `isPalindrome`. A number is a palindrome if its reversal is the same as itself. Write a test program that prompts the user to enter an integer and reports whether the integer is a palindrome.

5.4* (*Displaying an integer reversed*) Write the following method to display an integer in reverse order:

```
public static void reverse(int number)
```

For example, `reverse(3456)` displays `6543`. Write a test program that prompts the user to enter an integer and displays its reversal.

5.5* (*Sorting three numbers*) Write the following method to display three numbers in increasing order:

```
public static void displaySortedNumbers(
  double num1, double num2, double num3)
```

5.6* (*Displaying patterns*) Write a method to display a pattern as follows:

```
            1
          2 1
        3 2 1
...
n n-1 ... 3 2 1
```

The method header is

```
public static void displayPattern(int n)
```

5.7* (*Financial application: computing the future investment value*) Write a method that computes future investment value at a given interest rate for a specified number of years. The future investment is determined using the formula in Exercise 2.13.

Use the following method header:

```
public static double futureInvestmentValue(
  double investmentAmount, double monthlyInterestRate, int years)
```

For example, `futureInvestmentValue(10000, 0.05/12, 5)` returns `12833.59`.

Write a test program that prompts the user to enter the investment amount (e.g., 1000) and the interest rate (e.g., 9%) and prints a table that displays future value for the years from 1 to 30, as shown below:

```
The amount invested: 1000
Annual interest rate: 9%
Years       Future Value
1             1093.80
2             1196.41
...
29           13467.25
30           14730.57
```

5.8 (*Conversions between Celsius and Fahrenheit*) Write a class that contains the following two methods:

```
/** Converts from Celsius to Fahrenheit */
public static double celsiusToFahrenheit(double celsius)

/** Converts from Fahrenheit to Celsius */
public static double fahrenheitToCelsius(double fahrenheit)
```

The formula for the conversion is:

```
fahrenheit = (9.0 / 5) * celsius + 32
```

Write a test program that invokes these methods to display the following tables:

```
Celsius         Fahrenheit          Fahrenheit          Celsius

40.0            104.0               120.0               48.89
39.0            102.2               110.0               43.33
...
32.0            89.6                40.0                4.44
31.0            87.8                30.0                -1.11
```

5.9 (*Conversions between feet and meters*) Write a class that contains the following two methods:

```
/** Converts from feet to meters */
public static double footToMeter(double foot)

/** Converts from meters to feet */
public static double meterToFoot(double meter)
```

The formula for the conversion is:

```
meter = 0.305 * foot
```

Write a test program that invokes these methods to display the following tables:

```
Feet        Meters          Meters          Feet

1.0         0.305           20.0            65.574
2.0         0.61            25.0            81.967
...
9.0         2.745           60.0            196.721
10.0        3.05            65.0            213.115
```

5.10 (*Using the `isPrime` Method*) Listing 5.7, PrimeNumberMethod.java, provides the `isPrime(int number)` method for testing whether a number is prime. Use this method to find the number of prime numbers less than `10000`.

5.11 (*Financial application: computing commissions*) Write a method that computes the commission, using the scheme in Exercise 4.39. The header of the method is as follows:

```
public static double computeCommission(double salesAmount)
```

Write a test program that displays the following table:

```
Sales Amount      Commission

10000             900.0
15000             1500.0
...
95000             11100.0
100000            11700.0
```

5.12 (*Displaying characters*) Write a method that prints characters using the following header:

```
public static void printChars(char ch1, char ch2, int
  numberPerLine)
```

This method prints the characters between `ch1` and `ch2` with the specified numbers per line. Write a test program that prints ten characters per line from `'1'` to `'Z'`.

5.13* (*Summing series*) Write a method to compute the following series:

$$m(i) = \frac{1}{2} + \frac{2}{3} + \ldots + \frac{i}{i+1}$$

Write a test program that displays the following table:

```
i                 m(i)
1                 0.5000
2                 1.1667
...
19                16.4023
20                17.3546
```

5.14* (*Computing series*) Write a method to compute the following series:

$$m(i) = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \ldots + \frac{1}{2i-1} - \frac{1}{2i+1}\right)$$



Video Note
Compute π

Write a test program that displays the following table:

```
i             m(i)
10            3.04184
20            3.09162
...
90            3.13048
100           3.13159
```

5.15* (*Financial application: printing a tax table*) Listing 3.6 gives a program to compute tax. Write a method for computing tax using the following header:

```
public static double computetax(int status, double taxableIncome)
```

Use this method to write a program that prints a tax table for taxable income from $50,000 to $60,000 with intervals of $50 for all four statuses, as follows:

```
Taxable      Single     Married     Married      Head of
Income                  Joint       Separate     a House

50000        8688       6665        8688         7353
50050        8700       6673        8700         7365
...
59950        11175      8158        11175        9840
60000        11188      8165        11188        9853
```

5.16* (*Number of days in a year*) Write a method that returns the number of days in a year using the following header:

```
public static int numberOfDaysInAYear(int year)
```

Write a test program that displays the number of days in year from 2000 to 2010.

Sections 5.10–5.11

5.17* (*Displaying matrix of 0s and 1s*) Write a method that displays an n-by-n matrix using the following header:

```
public static void printMatrix(int n)
```

Each element is 0 or 1, which is generated randomly. Write a test program that prints a 3-by-3 matrix that may look like this:

```
0 1 0
0 0 0
1 1 1
```

5.18 (*Using the `Math.sqrt` method*) Write a program that prints the following table using the `sqrt` method in the `Math` class.

```
Number      SquareRoot

0               0.0000
2               1.4142
...
18              4.2426
20              4.4721
```

5.19* (*The `MyTriangle` class*) Create a class named `MyTriangle` that contains the following two methods:

```
/** Returns true if the sum of any two sides is
 *  greater than the third side. */
public static boolean isValid(
  double side1, double side2, double side3)

/** Returns the area of the triangle. */
public static double area(
  double side1, double side2, double side3)
```

Write a test program that reads three sides for a triangle and computes the area if the input is valid. Otherwise, it displays that the input is invalid. The formula for computing the area of a triangle is given in Exercise 2.21.

5.20 (*Using trigonometric methods*) Print the following table to display the `sin` value and `cos` value of degrees from 0 to 360 with increments of 10 degrees. Round the value to keep four digits after the decimal point.

```
Degree      Sin         Cos

0           0.0000      1.0000
10          0.1736      0.9848
...
350         -0.1736     0.9848
360         0.0000      1.0000
```

5.21** (*Statistics: computing mean and standard deviation*) In business applications, you are often asked to compute the mean and standard deviation of data. The mean is simply the average of the numbers. The standard deviation is a statistic that tells you how tightly all the various data are clustered around the mean in a set of data. For example, what is the average age of the students in a class? How close are the ages? If all the students are the same age, the deviation is 0. Write a program that prompts the user to enter ten numbers, and displays the mean and standard deviations of these numbers using the following formula:

$$mean = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{x_1 + x_2 + \ldots + x_n}{n} \qquad deviation = \sqrt{\frac{\sum_{i=1}^{n} x_i^2 - \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}}{n-1}}$$

Here is a sample run:

```
Enter ten numbers: 1 2 3 4.5 5.6 6 7 8 9 10 Enter
The mean is 5.61
The standard deviation is 2.99794
```



5.22** (*Math: approximating the square root*) Implement the `sqrt` method. The square root of a number, `num`, can be approximated by repeatedly performing a calculation using the following formula:

```
nextGuess = (lastGuess + (num / lastGuess)) / 2
```

When `nextGuess` and `lastGuess` are almost identical, `nextGuess` is the approximated square root.

The initial guess can be any positive value (e.g., `1`). This value will be the starting value for `lastGuess`. If the difference between `nextGuess` and `lastGuess` is less than a very small number, such as `0.0001`, you can claim that `nextGuess` is the approximated square root of `num`. If not, `nextGuess` becomes `lastGuess` and the approximation process continues.

Sections 5.10–5.11

5.23* (*Generating random characters*) Use the methods in `RandomCharacter` in Listing 5.10 to print 100 uppercase letters and then 100 single digits, printing ten per line.

5.24** (*Displaying current date and time*) Listing 2.9, ShowCurrentTime.java, displays the current time. Improve this example to display the current date and time. The calendar example in Listing 5.12, PrintCalendar.java, should give you some ideas on how to find year, month, and day.

5.25** (*Converting milliseconds to hours, minutes, and seconds*) Write a method that converts milliseconds to hours, minutes, and seconds using the following header:

```
public static String convertMillis(long millis)
```

The method returns a string as hours:minutes:seconds. For example, `convertMillis(5500)` returns a string 0:0:5, `convertMillis(100000)` returns a string 0:1:40, and `convertMillis(555550000)` returns a string 154:19:10.

Comprehensive

5.26** (*Palindromic prime*) A *palindromic prime* is a prime number and also palindromic. For example, 131 is a prime and also a palindromic prime. So are 313 and 757. Write a program that displays the first 100 palindromic prime numbers. Display 10 numbers per line and align the numbers properly, as follows:

```
  2    3    5    7   11  101  131  151  181  191
313  353  373  383  727  757  787  797  919  929
...
```

5.27** (*Emirp*) An *emirp* (prime spelled backward) is a nonpalindromic prime number whose reversal is also a prime. For example, 17 is a prime and 71 is a prime. So, 17 and 71 are emirps. Write a program that displays the first 100 emirps. Display 10 numbers per line and align the numbers properly, as follows:

```
 13   17   31   37   71   73   79   97  107  113
149  157  167  179  199  311  337  347  359  389
...
```

5.28** (*Mersenne prime*) A prime number is called a *Mersenne prime* if it can be written in the form $2^p - 1$ for some positive integer p. Write a program that finds all Mersenne primes with $p \leq 31$ and displays the output as follows:

```
p        2^p - 1
2          3
3          7
5         31
...
```

5.29** (*Game: craps*) Craps is a popular dice game played in casinos. Write a program to play a variation of the game, as follows:

Roll two dice. Each die has six faces representing values 1, 2, ..., and 6, respectively. Check the sum of the two dice. If the sum is 2, 3, or 12 (called *craps*), you lose; if the sum is 7 or 11 (called *natural*), you win; if the sum is another value (i.e., 4, 5, 6, 8, 9, or 10), a *point* is established. Continue to roll the dice until either a 7 or the same point value is rolled. If 7 is rolled, you lose. Otherwise, you win.

Your program acts as a single player. Here are some sample runs.



```
You rolled 5 + 6 = 11
You win
```



```
You rolled 1 + 2 = 3
You lose
```



```
You rolled 4 + 4 = 8
point is 8
You rolled 6 + 2 = 8
You win
```



```
You rolled 3 + 2 = 5
point is 5
You rolled 2 + 5 = 7
You lose
```

5.30** (*Twin primes*) Twin primes are a pair of prime numbers that differ by 2. For example, 3 and 5 are twin primes, 5 and 7 are twin primes, and 11 and 13 are twin primes. Write a program to find all twin primes less than 1000. Display the output as follows:

```
(3, 5)
(5, 7)
...
```

5.31** (*Financial: credit card number validation*) Credit card numbers follow certain patterns. A credit card number must have between 13 and 16 digits. It must start with:

- 4 for Visa cards
- 5 for Master cards
- 37 for American Express cards
- 6 for Discover cards

In 1954, Hans Luhn of IBM proposed an algorithm for validating credit card numbers. The algorithm is useful to determine whether a card number is entered correctly or whether a credit card is scanned correctly by a scanner. All credit card numbers are generated following this validity check, commonly known as the *Luhn check* or the *Mod 10 check*, which can be described as follows (for illustration, consider the card number 4388576018402626):

1. Double every second digit from right to left. If doubling of a digit results in a two-digit number, add up the two digits to get a single-digit number.

   2 * 2 = 4
   2 * 2 = 4
   4 * 2 = 8
   1 * 2 = 2
   6 * 2 = 12 (1 + 2 = 3)
   5 * 2 = 10 (1 + 0 = 1)
   8 * 2 = 16 (1 + 6 = 7)
   4 * 2 = 8

2. Now add all single-digit numbers from Step 1.

   4 + 4 + 8 + 2 + 3 + 1 + 7 + 8 = 37

3. Add all digits in the odd places from right to left in the card number.

   6 + 6 + 0 + 8 + 0 + 7 + 8 + 3 = 38

4. Sum the results from Step 2 and Step 3.

   37 + 38 = 75

5. If the result from Step 4 is divisible by 10, the card number is valid; otherwise, it is invalid. For example, the number 4388576018402626 is invalid, but the number 4388576018410707 is valid.

Write a program that prompts the user to enter a credit card number as a `long` integer. Display whether the number is valid or invalid. Design your program to use the following methods:

```
/** Return true if the card number is valid */
public static boolean isValid(long number)

/** Get the result from Step 2 */
public static int sumOfDoubleEvenPlace(long number)
```

```
/** Return this number if it is a single digit, otherwise, return
 * the sum of the two digits */
public static int getDigit(int number)

/** Return sum of odd place digits in number */
public static int sumOfOddPlace(long number)

/** Return true if the digit d is a prefix for number */
public static boolean prefixMatched(long number, int d)

/** Return the number of digits in d */
public static int getSize(long d)

/** Return the first k number of digits from number. If the
 * number of digits in number is less than k, return number. */
public static long getPrefix(long number, int k)
```

5.32** (*Game: chance of winning at craps*) Revise Exercise 5.29 to run it **10000** times and display the number of winning games.

5.33*** (*Current date and time*) Invoking **System.currentTimeMillis()** returns the elapse time in milliseconds since midnight of January 1, 1970. Write a program that displays the date and time. Here is a sample run:



```
Current date and time is May 16, 2009 10:34:23
```

5.34** (*Printing calendar*) Exercise 3.21 uses Zeller's congruence to calculate the day of the week. Simplify Listing 5.12, PrintCalendar.java, using Zeller's algorithm to get the start day of the month.

5.35 (*Geometry: area of a pentagon*) The area of a pentagon can be computed using the following formula:

$$Area = \frac{5 \times s^2}{4 \times \tan\left(\frac{\pi}{5}\right)}$$

Write a program that prompts the user to enter the side of a pentagon and displays its area.

5.36* (*Geometry: area of a regular polygon*) A regular polygon is an n-sided polygon in which all sides are of the same length and all angles have the same degree (i.e., the polygon is both equilateral and equiangular). The formula for computing the area of a regular polygon is

$$Area = \frac{n \times s^2}{4 \times \tan\left(\frac{\pi}{n}\right)}.$$

Write a method that returns the area of a regular polygon using the following header:

```
public static double area(int n, double side)
```

Write a main method that prompts the user to enter the number of sides and the side of a regular polygon and displays its area.

CHAPTER 6

SINGLE-DIMENSIONAL ARRAYS

Objectives

- To describe why arrays are necessary in programming (§6.1).
- To declare array reference variables and create arrays (§§6.2.1–6.2.2).
- To initialize the values in an array (§6.2.3).
- To access array elements using indexed variables (§6.2.4).
- To declare, create, and initialize an array using an array initializer (§6.2.5).
- To program common array operations (displaying arrays, summing all elements, finding min and max elements, random shuffling, shifting elements) (§6.2.6).
- To simplify programming using the for-each loops (§6.2.7).
- To apply arrays in the `LottoNumbers` and `DeckOfCards` problems (§§6.3–6.4).
- To copy contents from one array to another (§6.5).
- To develop and invoke methods with array arguments and return value (§6.6–6.7).
- To define a method with variable-length argument list (§6.8).
- To search elements using the linear (§6.9.1) or binary (§6.9.2) search algorithm.
- To sort an array using the selection sort (§6.10.1)
- To sort an array using the insertion sort (§6.10.2).
- To use the methods in the `Arrays` class (§6.11).



6.1 Introduction

problem

Often you will have to store a large number of values during the execution of a program. Suppose, for instance, that you need to read 100 numbers, compute their average, and find out how many numbers are above the average. Your program first reads the numbers and computes their average, then compares each number with the average to determine whether it is above the average. In order to accomplish this task, the numbers must all be stored in variables. You have to declare 100 variables and repeatedly write almost identical code 100 times.

why array?

Writing a program this way would be impractical. So, how do you solve this problem?

what is array?

An efficient, organized approach is needed. Java and most other high-level languages provide a data structure, the *array*, which stores a fixed-size sequential collection of elements of the same type. In the present case, you can store all 100 numbers into an array and access them through a single array variable. The solution may look like this:

declare array

store number in array

get average

above average?

```
1 public class AnalyzeNumbers {
2   public static void main(String[] args) {
3     final int NUMBER_OF_ELEMENTS = 100;
4     double[] numbers = new double[NUMBER_OF_ELEMENTS];
5     double sum = 0;
6
7     java.util.Scanner input = new java.util.Scanner(System.in);
8     for (int i = 0; i < NUMBER_OF_ELEMENTS; i++) {
9       System.out.print("Enter a new number: ");
10      numbers[i] = input.nextDouble();
11      sum += numbers[i];
12    }
13
14    double average = sum / NUMBER_OF_ELEMENTS;
15
16    int count = 0; // The number of elements above average
17    for (int i = 0; i < NUMBER_OF_ELEMENTS; i++)
18      if (numbers[i] > average)
19        count++;
20
21    System.out.println("Average is " + average);
22    System.out.println("Number of elements above the average "
23      + count);
24  }
25 }
```

The program creates an array of 100 elements in line 4, stores numbers into the array in line 10, adds each number to sum in line 11, and obtains the average in line 14. It then compares each number in the array with the average to count the number of values above the average (lines 16–19).

This chapter introduces single-dimensional arrays. The next chapter will introduce two-dimensional and multidimensional arrays.

6.2 Array Basics

An array is used to store a collection of data, but often we find it more useful to think of an array as a collection of variables of the same type. Instead of declaring individual variables, such as number0, number1, . . . , and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], . . . , and numbers[99] to represent individual variables. This section introduces how to declare array variables, create arrays, and process arrays using indexed variables.

6.2.1 Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array and specify the array's *element type*. Here is the syntax for declaring an array variable: element type

```
elementType[] arrayRefVar;
```

The `elementType` can be any data type, and all elements in the array will have the same data type. For example, the following code declares a variable `myList` that references an array of double elements.

```
double[] myList;
```



Note

You can also use `elementType arrayRefVar[]` to declare an array variable. This style comes from the C language and was adopted in Java to accommodate C programmers. The style `elementType[] arrayRefVar` is preferred. preferred syntax

6.2.2 Creating Arrays

Unlike declarations for primitive data type variables, the declaration of an array variable does not allocate any space in memory for the array. It creates only a storage location for the reference to an array. If a variable does not contain a reference to an array, the value of the variable is `null`. You cannot assign elements to an array unless it has already been created. After an array variable is declared, you can create an array by using the `new` operator with the following syntax:

```
arrayRefVar = new elementType[arraySize];
```

new operator

This statement does two things: (1) it creates an array using `new elementType[arraySize]`; (2) it assigns the reference of the newly created array to the variable `arrayRefVar`.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

```
elementType arrayRefVar = new elementType[arraySize];
```

or

```
elementType arrayRefVar[] = new elementType[arraySize];
```

Here is an example of such a statement:

```
double[] myList = new double[10];
```

This statement declares an array variable, `myList`, creates an array of ten elements of `double` type, and assigns its reference to `myList`. To assign values to the elements, use the syntax:

```
arrayRefVar[index] = value;
```

For example, the following code initializes the array.

```
myList[0] = 5.6;
myList[1] = 4.5;
myList[2] = 3.3;
myList[3] = 13.2;
```

```
myList[4] = 4.0;
myList[5] = 34.33;
myList[6] = 34.0;
myList[7] = 45.45;
myList[8] = 99.993;
myList[9] = 11123;
```

The array is pictured in Figure 6.1.

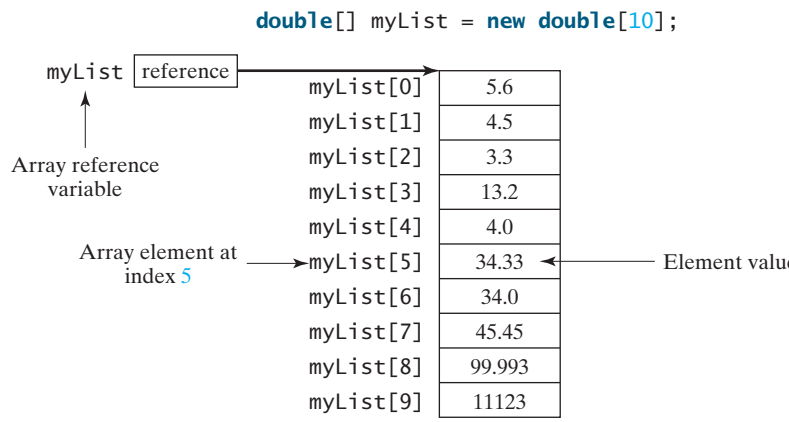


FIGURE 6.1 The array `myList` has ten elements of `double` type and `int` indices from `0` to `9`.

array vs. array variable

Note

An array variable that appears to hold an array actually contains a reference to that array. Strictly speaking, an array variable and an array are different, but most of the time the distinction can be ignored. Thus it is all right to say, for simplicity, that `myList` is an array, instead of stating, at greater length, that `myList` is a variable that contains a reference to an array of ten double elements.

6.2.3 Array Size and Default Values

When space for an array is allocated, the array size must be given, specifying the number of elements that can be stored in it. The size of an array cannot be changed after the array is created. Size can be obtained using `arrayRefVar.length`. For example, `myList.length` is `10`.

array length

When an array is created, its elements are assigned the default value of `0` for the numeric primitive data types, `'\u0000'` for `char` types, and `false` for `boolean` types.

default values

6.2.4 Array Indexed Variables

0 based

The array elements are accessed through the index. Array indices are `0` based; that is, they range from `0` to `arrayRefVar.length-1`. In the example in Figure 6.1, `myList` holds ten `double` values, and the indices are from `0` to `9`.

indexed variables

Each element in the array is represented using the following syntax, known as an *indexed variable*:

```
arrayRefVar[index];
```

For example, `myList[9]` represents the last element in the array `myList`.

Caution

Some languages use parentheses to reference an array element, as in `myList(9)`. But Java uses brackets, as in `myList[9]`.

After an array is created, an indexed variable can be used in the same way as a regular variable. For example, the following code adds the values in `myList[0]` and `myList[1]` to `myList[2]`.

```
myList[2] = myList[0] + myList[1];
```

The following loop assigns `0` to `myList[0]`, `1` to `myList[1]`, ..., and `9` to `myList[9]`:

```
for (int i = 0; i < myList.length; i++) {
  myList[i] = i;
}
```

6.2.5 Array Initializers

Java has a shorthand notation, known as the *array initializer*, which combines in one statement declaring an array, creating an array, and initializing, using the following syntax:

```
elementType[] arrayRefVar = {value0, value1, ..., valuek};
```

For example,

```
double[] myList = {1.9, 2.9, 3.4, 3.5};
```

This statement declares, creates, and initializes the array `myList` with four elements, which is equivalent to the statements shown below:

```
double[] myList = new double[4];
myList[0] = 1.9;
myList[1] = 2.9;
myList[2] = 3.4;
myList[3] = 3.5;
```



Caution

The `new` operator is not used in the array-initializer syntax. Using an array initializer, you have to declare, create, and initialize the array all in one statement. Splitting it would cause a syntax error. Thus the next statement is wrong:

```
double[] myList;
myList = {1.9, 2.9, 3.4, 3.5};
```

6.2.6 Processing Arrays

When processing array elements, you will often use a `for` loop—for two reasons:

- All of the elements in an array are of the same type. They are evenly processed in the same fashion repeatedly using a loop.
- Since the size of the array is known, it is natural to use a `for` loop.

Assume the array is created as follows:

```
double[] myList = new double[10];
```

Here are some examples of processing arrays:

1. (*Initializing arrays with input values*) The following loop initializes the array `myList` with user input values.

```
java.util.Scanner input = new java.util.Scanner(System.in);
System.out.print("Enter " + myList.length + " values: ");
for (int i = 0; i < myList.length; i++)
  myList[i] = input.nextDouble();
```

2. (*Initializing arrays with random values*) The following loop initializes the array `myList` with random values between `0.0` and `100.0`, but less than `100.0`.

```
for (int i = 0; i < myList.length; i++) {
  myList[i] = Math.random() * 100;
}
```

3. (*Displaying arrays*) To print an array, you have to print each element in the array using a loop like the following:

```
for (int i = 0; i < myList.length; i++) {
  System.out.print(myList[i] + " ");
}
```



Tip

print character array

For an array of the `char[]` type, it can be printed using one print statement. For example, the following code displays `Dallas`:

```
char[] city = {'D', 'a', 'l', 'l', 'a', 's'};
System.out.println(city);
```

4. (*Summing all elements*) Use a variable named `total` to store the sum. Initially `total` is `0`. Add each element in the array to `total` using a loop like this:

```
double total = 0;
for (int i = 0; i < myList.length; i++) {
  total += myList[i];
}
```

5. (*Finding the largest element*) Use a variable named `max` to store the largest element. Initially `max` is `myList[0]`. To find the largest element in the array `myList`, compare each element with `max`, and update `max` if the element is greater than `max`.

```
double max = myList[0];
for (int i = 1; i < myList.length; i++) {
  if (myList[i] > max) max = myList[i];
}
```

6. (*Finding the smallest index of the largest element*) Often you need to locate the largest element in an array. If an array has more than one largest element, find the smallest index of such an element. Suppose the array `myList` is {`1`, `5`, `3`, `4`, `5`, `5`}. The largest element is `5` and the smallest index for `5` is `1`. Use a variable named `max` to store the largest element and a variable named `indexOfMax` to denote the index of the largest element. Initially `max` is `myList[0]`, and `indexOfMax` is `0`. Compare each element in `myList` with `max`, and update `max` and `indexOfMax` if the element is greater than `max`.

```
double max = myList[0];
int indexOfMax = 0;
for (int i = 1; i < myList.length; i++) {
  if (myList[i] > max) {
    max = myList[i];
    indexOfMax = i;
  }
}
```

What is the consequence if `(myList[i] > max)` is replaced by `(myList[i] >= max)`?

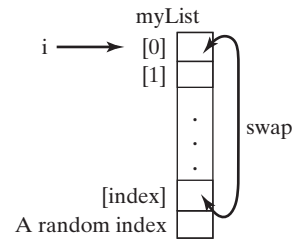


Video Note
Random shuffling

7. (*Random shuffling*) In many applications, you need to randomly reorder the elements in an array. This is called a *shuffling*. To accomplish this, for each element `myList[i]`, randomly generate an index `j` and swap `myList[i]` with `myList[j]`, as follows:

```
for (int i = 0; i < myList.length; i++) {
  // Generate an index j randomly
  int index = (int) (Math.random()
    * mylist.length);

  // Swap myList[i] with myList[j]
  double temp = myList[i];
  myList[i] = myList[index]
  myList[index] = temp;
}
```

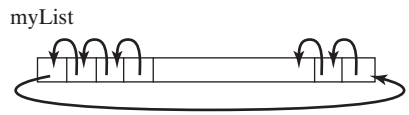


8. (*Shifting elements*) Sometimes you need to shift the elements left or right. Here is an example to shift the elements one position to the left and fill the last element with the first element:

```
double temp = myList[0]; // Retain the first element

// Shift elements left
for (int i = 1; i < myList.length; i++) {
  myList[i - 1] = myList[i];
}

// Move the first element to fill in the last position
myList[myList.length - 1] = temp;
```



6.2.7 For-each Loops

Java supports a convenient **for** loop, known as a *for-each loop* or *enhanced for loop*, which enables you to traverse the array sequentially without using an index variable. For example, the following code displays all the elements in the array **myList**:

```
for (double u: myList) {
  System.out.println(u);
}
```

You can read the code as "for each element **u** in **myList** do the following." Note that the variable, **u**, must be declared the same type as the elements in **myList**.

In general, the syntax for a **for-each** loop is

```
for (elementType element: arrayRefVar) {
  // Process the element
}
```

You still have to use an index variable if you wish to traverse the array in a different order or change the elements in the array.



Caution

Accessing an array out of bounds is a common programming error that throws a runtime **ArrayIndexOutOfBoundsException**. To avoid it, make sure that you do not use an index beyond **arrayRefVar.length – 1**.

ArrayIndexOutOfBounds-Exception

Programmers often mistakenly reference the first element in an array with index **1**, but it should be **0**. This is called the *off-by-one error*. It is a common error in a loop to use **<=** where **<** should be used. For example, the following loop is wrong,

off-by-one error

```
for (int i = 0; i <= list.length; i++)
  System.out.print(list[i] + " ");
```

The **<=** should be replaced by **<**.



Video Note
Lotto numbers

6.3 Problem: Lotto Numbers

Each ticket for the Pick-10 lotto has **10** unique numbers ranging from **1** to **99**. Suppose you buys a lot of tickets and like to have them cover all numbers from **1** to **99**. Write a program that reads the ticket numbers from a file and checks whether all numbers are covered. Assume the last number in the file is **0**. Suppose the file contains the numbers

```
80 3 87 62 30 90 10 21 46 27
12 40 83 9 39 88 95 59 20 37
80 40 87 67 31 90 11 24 56 77
11 48 51 42 8 74 1 41 36 53
52 82 16 72 19 70 44 56 29 33
54 64 99 14 23 22 94 79 55 2
60 86 34 4 31 63 84 89 7 78
43 93 97 45 25 38 28 26 85 49
47 65 57 67 73 69 32 71 24 66
92 98 96 77 6 75 17 61 58 13
35 81 18 15 5 68 91 50 76
0
```

Your program should display

```
The tickets cover all numbers
```

Suppose the file contains the numbers

```
11 48 51 42 8 74 1 41 36 53
52 82 16 72 19 70 44 56 29 33
0
```

Your program should display

```
The tickets don't cover all numbers
```

How do you mark a number as covered? You can create an array with **99** **boolean** elements. Each element in the array can be used to mark whether a number is covered. Let the array be **isCovered**. Initially, each element is **false**, as shown in Figure 6.2(a). Whenever a number is read, its corresponding element is set to **true**. Suppose the numbers entered are **1**, **2**, **3**, **99**, **0**. When number **1** is read, **isCovered[0]** is set to **true** (see Figure 6.2(b)). When number **2** is read, **isCovered[2 - 1]** is set to **true** (see Figure 6.2(c)). When number **3** is read,

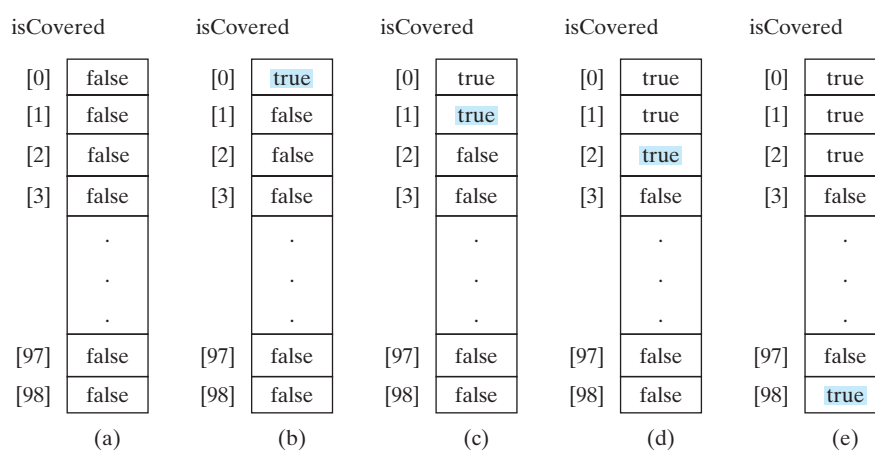


FIGURE 6.2 If number **i** appears in a Lotto ticket, **isCovered[i-1]** is set to true.

`isCovered[3 - 1]` is set to `true` (see Figure 6.2(d)). When number `99` is read, set `isCovered[98]` to `true` (see Figure 6.2(e)).

The algorithm for the program can be described as follows:

```
for each number k read from the file,
  mark number k as covered by setting isCovered[k - 1] true;

if every isCovered[i] is true
  The tickets cover all numbers
else
  The tickets don't cover all numbers
```

The complete program is given in Listing 6.1.

LISTING 6.1 LottoNumbers.java

```
1  import java.util.Scanner;
2
3  public class LottoNumbers {
4    public static void main(String args[]) {
5      Scanner input = new Scanner(System.in);
6      boolean[] isCovered = new boolean[99]; // Default is false          create and initialize array
7
8      // Read each number and mark its corresponding element covered
9      int number = input.nextInt();                                       read number
10     while (number != 0) {
11       isCovered[number - 1] = true;                                     mark number covered
12       number = input.nextInt();                                         read number
13     }
14
15     // Check whether all covered
16     boolean allCovered = true; // Assume all covered initially
17     for (int i = 0; i < 99; i++)
18       if (!isCovered[i]) {
19         allCovered = false; // Find one number not covered
20         break;
21       }
22
23     // Display result
24     if (allCovered)                                                     check allCovered?
25       System.out.println("The tickets cover all numbers");
26     else
27       System.out.println("The tickets don't cover all numbers");
28   }
29 }
```

Suppose you have created a text file named LottoNumbers.txt that contains the input data `2 5 6 5 4 3 23 43 2 0`. You can run the program using the following command:

```
java LottoNumbers < LottoNumbers.txt
```

The program can be traced as follows:

The program creates an array of `99 boolean` elements and initializes each element to `false` (line 6). It reads the first number from the file (line 9). The program then repeats the following operations in a loop:

- If the number is not zero, set its corresponding value in array `isCovered` to `true` (line 11);
- Read the next number (line 12).



line	Representative elements in array isCovered							number	allCovered
	[1]	[2]	[3]	[4]	[5]	[22]	[42]		
6	false	false	false	false	false	false	false		
9								2	
11	true								
12								5	
11				true					
12								6	
11					true				
12								5	
11				true					
12								4	
11			true						
12								3	
11		true							
12								23	
11						true			
12								43	
11							true		
12								2	
11	true								
12								0	
16									true
18(i=0)									false

When the input is **0**, the input ends. The program checks whether all numbers are covered in lines 16–21 and displays the result in lines 24–27.

6.4 Problem: Deck of Cards

The problem is to write a program that picks four cards randomly from a deck of **52** cards. All the cards can be represented using an array named **deck**, filled with initial values **0** to **51**, as follows:

```
int[] deck = new int[52];

// Initialize cards
for (int i = 0; i < deck.length; i++)
  deck[i] = i;
```

Card numbers **0** to **12**, **13** to **25**, **26** to **38**, **39** to **51** represent 13 Spades, 13 Hearts, 13 Diamonds, and 13 Clubs, respectively, as shown in Figure 6.3. After shuffling the array **deck**, pick the first four cards from **deck**. **cardNumber / 13** determines the suit of the card and **cardNumber % 13** determines the rank of the card.

Listing 6.2 gives the solution to the problem.

LISTING 6.2 DeckOfCards.java

```
1 public class DeckOfCards {
2   public static void main(String[] args) {
3     int[] deck = new int[52];                                  create array deck
4     String[] suits = {"Spades", "Hearts", "Diamonds", "Clubs"};  array of strings
5     String[] ranks = {"Ace", "2", "3", "4", "5", "6", "7", "8", "9",  array of strings
6       "10", "Jack", "Queen", "King"};
7
8     // Initialize cards
9     for (int i = 0; i < deck.length; i++)                      initialize deck
10      deck[i] = i;
11
12    // Shuffle the cards
13    for (int i = 0; i < deck.length; i++) {                    shuffle deck
14      // Generate an index randomly
15      int index = (int)(Math.random() * deck.length);
16      int temp = deck[i];
17      deck[i] = deck[index];
18      deck[index] = temp;
19    }
20
21    // Display the first four cards
22    for (int i = 0; i < 4; i++) {
23      String suit = suits[deck[i] / 13];                       suit of a card
24      String rank = ranks[deck[i] % 13];                       rank of a card
25      System.out.println("Card number " + deck[i] + ": "
26        + rank + " of " + suit);
27    }
28  }
29 }
```

```
Card number 6: 7 of Spades
Card number 48: 10 of Clubs
Card number 11: Queen of Spades
Card number 24: Queen of Hearts
```

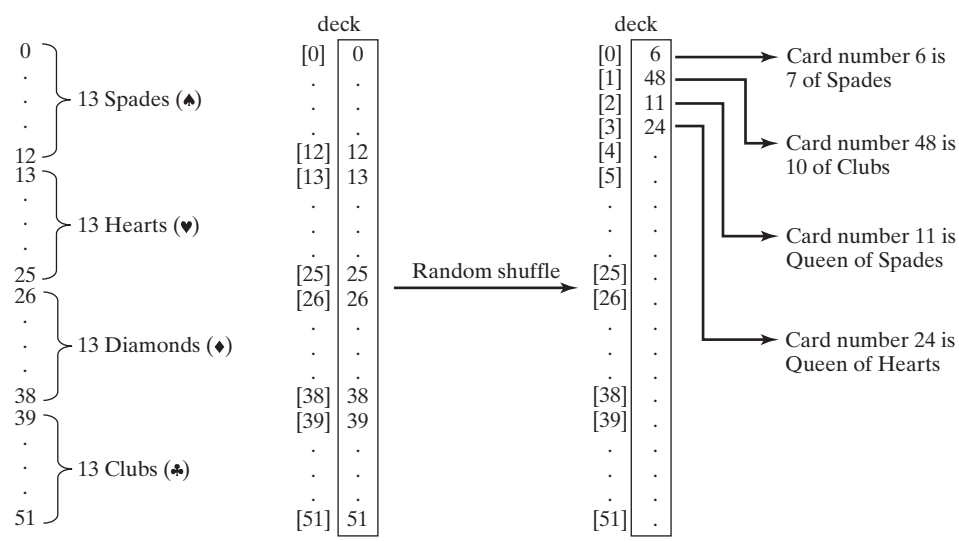


FIGURE 6.3 52 cards are stored in an array named **deck**.

The program defines an array `suits` for four suits (line 4) and an array `ranks` for 13 cards in a suits (lines 5–6). Each element in these arrays is a string.

The `deck` is initialized with values `0` to `51` in lines 9–10. A deck value `0` represents card Ace of Spades, `1` represents card 2 of Spades, `13` represents card Ace of Hearts, `14` represents card 2 of Hearts.

Lines 13–19 randomly shuffle the deck. After a deck is shuffled, `deck[i]` contains an arbitrary value. `deck[i] / 13` is `0`, `1`, `2`, or `3`, which determines a suit (line 23). `deck[i] % 13` is a value between `0` and `12`, which determines a rank (line 24).

6.5 Copying Arrays

Often, in a program, you need to duplicate an array or a part of an array. In such cases you could attempt to use the assignment statement (`=`), as follows:

copy reference

```
list2 = list1;
```

This statement does not copy the contents of the array referenced by `list1` to `list2`, but merely copies the reference value from `list1` to `list2`. After this statement, `list1` and `list2` reference to the same array, as shown in Figure 6.4. The array previously referenced by `list2` is no longer referenced; it becomes garbage, which will be automatically collected by the Java Virtual Machine.

garbage collection

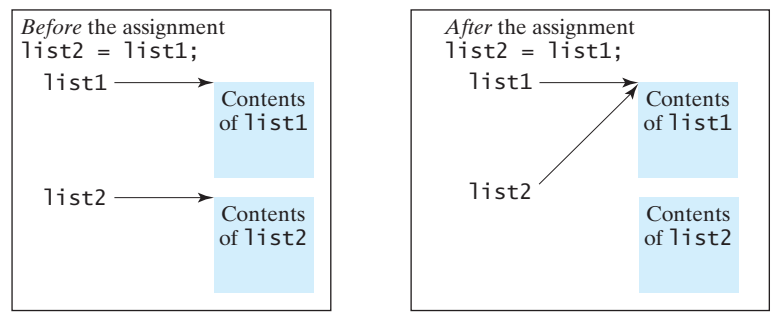


FIGURE 6.4 Before the assignment statement, `list1` and `list2` point to separate memory locations. After the assignment, the reference of the `list1` array is passed to `list2`.

In Java, you can use assignment statements to copy primitive data type variables, but not arrays. Assigning one array variable to another array variable actually copies one reference to another and makes both variables point to the same memory location.

There are three ways to copy arrays:

- Use a loop to copy individual elements one by one.
- Use the static `arraycopy` method in the `System` class.
- Use the `clone` method to copy arrays; this will be introduced in Chapter 14, "Abstract Classes and Interfaces."

You can write a loop to copy every element from the source array to the corresponding element in the target array. The following code, for instance, copies `sourceArray` to `targetArray` using a `for` loop.

```
int[] sourceArray = {2, 3, 1, 5, 10};
int[] targetArray = new int[sourceArray.length];
```

```
for (int i = 0; i < sourceArray.length; i++) {
  targetArray[i] = sourceArray[i];
}
```

Another approach is to use the `arraycopy` method in the `java.lang.System` class to copy arrays instead of using a loop. The syntax for `arraycopy` is shown below:

```
arraycopy(sourceArray, src_pos, targetArray, tar_pos, length);
```

arraycopy method

The parameters `src_pos` and `tar_pos` indicate the starting positions in `sourceArray` and `targetArray`, respectively. The number of elements copied from `sourceArray` to `targetArray` is indicated by `length`. For example, you can rewrite the loop using the following statement:

```
System.arraycopy(sourceArray, 0, targetArray, 0, sourceArray.length);
```

The `arraycopy` method does not allocate memory space for the target array. The target array must have already been created with its memory space allocated. After the copying takes place, `targetArray` and `sourceArray` have the same content but independent memory locations.



Note

The `arraycopy` method violates the Java naming convention. By convention, this method should be named `arrayCopy` (i.e., with an uppercase C).

6.6 Passing Arrays to Methods

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an `int` array:

```
public static void printArray(int[] array) {
  for (int i = 0; i < array.length; i++) {
    System.out.print(array[i] + " ");
  }
}
```

You can invoke it by passing an array. For example, the following statement invokes the `printArray` method to display `3`, `1`, `2`, `6`, `4`, and `2`.

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```



Note

The preceding statement creates an array using the following syntax:

```
new elementType[]{value0, value1, ..., valuek};
```

There is no explicit reference variable for the array. Such array is called an *anonymous array*.

anonymous arrays

Java uses *pass-by-value* to pass arguments to a method. There are important differences between passing the values of variables of primitive data types and passing arrays.

pass-by-value

- For an argument of a primitive type, the argument's value is passed.
- For an argument of an array type, the value of the argument is a reference to an array; this reference value is passed to the method. Semantically, it can be best described as *pass-by-sharing*, i.e., the array in the method is the same as the array being passed. So if you change the array in the method, you will see the change outside the method.

pass-by-sharing

Take the following code, for example:

```
public class Test {
  public static void main(String[] args) {
    int x = 1; // x represents an int value
    int[] y = new int[10]; // y represents an array of int values

    m(x, y); // Invoke m with arguments x and y

    System.out.println("x is " + x);
    System.out.println("y[0] is " + y[0]);
  }

  public static void m(int number, int[] numbers) {
    number = 1001; // Assign a new value to number
    numbers[0] = 5555; // Assign a new value to numbers[0]
  }
}
```



```
x is 1
y[0] is 5555
```

You will see that after `m` is invoked, `x` remains `1`, but `y[0]` is `5555`. This is because `y` and `numbers`, although they are independent variables, reference to the same array, as illustrated in Figure 6.5. When `m(x, y)` is invoked, the values of `x` and `y` are passed to `number` and `numbers`. Since `y` contains the reference value to the array, `numbers` now contains the same reference value to the same array.

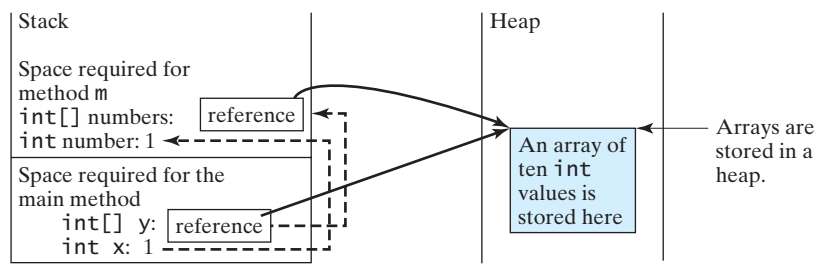


FIGURE 6.5 The primitive type value in `x` is passed to `number`, and the reference value in `y` is passed to `numbers`.

heap



Note

The JVM stores the array in an area of memory called the *heap*, which is used for dynamic memory allocation where blocks of memory are allocated and freed in an arbitrary order.

6.6.1 Passing Array Arguments

Listing 6.3 gives another program that shows the difference between passing a primitive data type value and an array reference variable to a method.

The program contains two methods for swapping elements in an array. The first method, named `swap`, fails to swap two `int` arguments. The second method, named `swapFirstTwoInArray`, successfully swaps the first two elements in the array argument.

LISTING 6.3 TestPassArray.java

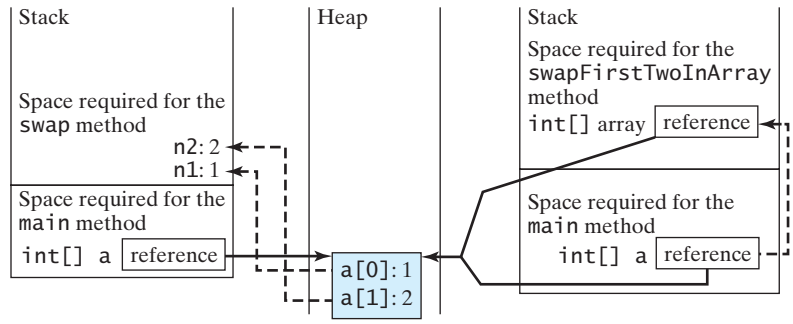
```
1 public class TestPassArray {
2   /** Main method */
3   public static void main(String[] args) {
4     int[] a = {1, 2};
5
6     // Swap elements using the swap method
7     System.out.println("Before invoking swap");
8     System.out.println("array is {" + a[0] + ", " + a[1] + "}");
9     swap(a[0], a[1]);                                          false swap
10    System.out.println("After invoking swap");
11    System.out.println("array is {" + a[0] + ", " + a[1] + "}");
12
13    // Swap elements using the swapFirstTwoInArray method
14    System.out.println("Before invoking swapFirstTwoInArray");
15    System.out.println("array is {" + a[0] + ", " + a[1] + "}");
16    swapFirstTwoInArray(a);                                    swap array elements
17    System.out.println("After invoking swapFirstTwoInArray");
18    System.out.println("array is {" + a[0] + ", " + a[1] + "}");
19  }
20
21  /** Swap two variables */
22  public static void swap(int n1, int n2) {
23    int temp = n1;
24    n1 = n2;
25    n2 = temp;
26  }
27
28  /** Swap the first two elements in the array */
29  public static void swapFirstTwoInArray(int[] array) {
30    int temp = array[0];
31    array[0] = array[1];
32    array[1] = temp;
33  }
34 }
```

```
Before invoking swap
array is {1, 2}
After invoking swap
array is {1, 2}
Before invoking swapFirstTwoInArray
array is {1, 2}
After invoking swapFirstTwoInArray
array is {2, 1}
```



As shown in Figure 6.6, the two elements are not swapped using the `swap` method. However, they are swapped using the `swapFirstTwoInArray` method. Since the parameters in the `swap` method are primitive type, the values of `a[0]` and `a[1]` are passed to `n1` and `n2` inside the method when invoking `swap(a[0], a[1])`. The memory locations for `n1` and `n2` are independent of the ones for `a[0]` and `a[1]`. The contents of the array are not affected by this call.

The parameter in the `swapFirstTwoInArray` method is an array. As shown in Figure 6.6, the reference of the array is passed to the method. Thus the variables `a` (outside the method) and `array` (inside the method) both refer to the same array in the same memory location. Therefore, swapping `array[0]` with `array[1]` inside the method `swapFirstTwoInArray` is the same as swapping `a[0]` with `a[1]` outside of the method.



Invoke `swap(int n1, int n2)`. The primitive type values in `a[0]` and `a[1]` are passed to the `swap` method.

The arrays are stored in a heap.

Invoke `swapFirstTwoInArray(int[] array)`. The reference value in `a` is passed to the `swapFirstTwoInArray` method.

FIGURE 6.6 When passing an array to a method, the reference of the array is passed to the method.

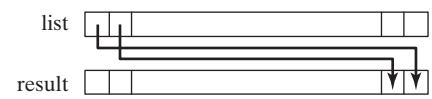
6.7 Returning an Array from a Method

You can pass arrays when invoking a method. A method may also return an array. For example, the method shown below returns an array that is the reversal of another array:

```
1  public static int[] reverse(int[] list) {
2    int[] result = new int[list.length];
3
4    for (int i = 0, j = result.length - 1;
5         i < list.length; i++, j--) {
6      result[j] = list[i];
7    }
8
9    return result;
10 }
```

create array

return array



Line 2 creates a new array **`result`**. Lines 4–7 copy elements from array **`list`** to array **`result`**. Line 9 returns the array. For example, the following statement returns a new array **`list2`** with elements **`6`**, **`5`**, **`4`**, **`3`**, **`2`**, **`1`**.

```
int[] list1 = {1, 2, 3, 4, 5, 6};
int[] list2 = reverse(list1);
```

6.7.1 Case Study: Counting the Occurrences of Each Letter

Listing 6.4 presents a program to count the occurrences of each letter in an array of characters. The program does the following:

1. Generate **`100`** lowercase letters randomly and assign them to an array of characters, as shown in Figure 6.7(a). You can obtain a random letter by using the **`getRandomLowerCaseLetter()`** method in the **`RandomCharacter`** class in Listing 5.10

2. Count the occurrences of each letter in the array. To do so, create an array, say **`counts`**, of **`26`** **`int`** values, each of which counts the occurrences of a letter, as shown in Figure 6.7(b). That is, **`counts[0]`** counts the number of **`a`**'s, **`counts[1]`** counts the number of **`b`**'s, and so on.

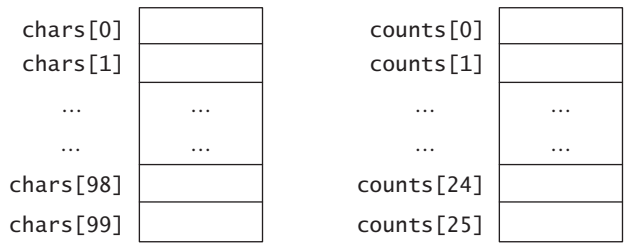


FIGURE 6.7 The `chars` array stores `100` characters, and the `counts` array stores `26` counts, each of which counts the occurrences of a letter.

LISTING 6.4 CountLettersInArray.java

```
1  public class CountLettersInArray {
2    /** Main method */
3    public static void main(String[] args) {
4      // Declare and create an array
5      char[] chars = createArray();                          create array
6
7      // Display the array
8      System.out.println("The lowercase letters are:");
9      displayArray(chars);                                   pass array
10
11     // Count the occurrences of each letter
12     int[] counts = countLetters(chars);                    return array
13
14     // Display counts
15     System.out.println();
16     System.out.println("The occurrences of each letter are:");
17     displayCounts(counts);                                 pass array
18   }
19
20   /** Create an array of characters */
21   public static char[] createArray() {
22     // Declare an array of characters and create it
23     char[] chars = new char[100];
24
25     // Create lowercase letters randomly and assign
26     // them to the array
27     for (int i = 0; i < chars.length; i++)
28       chars[i] = RandomCharacter.getRandomLowerCaseLetter();
29
30     // Return the array
31     return chars;
32   }
33
34   /** Display the array of characters */
35   public static void displayArray(char[] chars) {
36     // Display the characters in the array 20 on each line
37     for (int i = 0; i < chars.length; i++) {
38       if ((i + 1) % 20 == 0)
39         System.out.println(chars[i]);
```

```
40        else
41          System.out.print(chars[i] + " ");
42      }
43    }
44
45    /** Count the occurrences of each letter */
46    public static int[] countLetters(char[] chars) {
47      // Declare and create an array of 26 int
48      int[] counts = new int[26];
49
50      // For each lowercase letter in the array, count it
51      for (int i = 0; i < chars.length; i++)
52        counts[chars[i] - 'a']++;
53
54      return counts;
55    }
56
57    /** Display counts */
58    public static void displayCounts(int[] counts) {
59      for (int i = 0; i < counts.length; i++) {
60        if ((i + 1) % 10 == 0)
61          System.out.println(counts[i] + " " + (char)(i + 'a'));
62        else
63          System.out.print(counts[i] + " " + (char)(i + 'a') + " ");
64      }
65    }
66  }
```

count



```
The lowercase letters are:
e y l s r i b k j v j h a b z n w b t v
s c c k r d w a m p w v u n q a m p l o
a z g d e g f i n d x m z o u l o z j v
h w i w n t g x w c d o t x h y v z y z
q e a m f w p g u q t r e n n w f c r f

The occurrences of each letter are:
5 a 3 b 4 c 4 d 4 e 4 f 4 g 3 h 3 i 3 j
2 k 3 l 4 m 6 n 4 o 3 p 3 q 4 r 2 s 4 t
3 u 5 v 8 w 3 x 3 y 6 z
```

The **createArray** method (lines 21–32) generates an array of **100** random lowercase letters. Line 5 invokes the method and assigns the array to **chars**. What would be wrong if you rewrote the code as follows?

```
char[] chars = new char[100];
chars = createArray();
```

You would be creating two arrays. The first line would create an array by using **new char[100]**. The second line would create an array by invoking **createArray()** and assign the reference of the array to **chars**. The array created in the first line would be garbage because it is no longer referenced. Java automatically collects garbage behind the scenes. Your program would compile and run correctly, but it would create an array unnecessarily.

Invoking **getRandomLowerCaseLetter()** (line 28) returns a random lowercase letter. This method is defined in the **RandomCharacter** class in Listing 5.10.

The **countLetters** method (lines 46–55) returns an array of **26 int** values, each of which stores the number of occurrences of a letter. The method processes each letter in the

array and increases its count by one. A brute-force approach to count the occurrences of each letter might be as follows:

```
for (int i = 0; i < chars.length; i++)
  if (chars[i] == 'a')
    counts[0]++;
  else if (chars[i] == 'b')
    counts[1]++;
  ...
```

But a better solution is given in lines 51–52.

```
for (int i = 0; i < chars.length; i++)
  counts[chars[i] - 'a']++;
```

If the letter (`chars[i]`) is `'a'`, the corresponding count is `counts['a' - 'a']` (i.e., `counts[0]`). If the letter is `'b'`, the corresponding count is `counts['b' - 'a']` (i.e., `counts[1]`), since the Unicode of `'b'` is one more than that of `'a'`. If the letter is `'z'`, the corresponding count is `counts['z' - 'a']` (i.e., `counts[25]`), since the Unicode of `'z'` is `25` more than that of `'a'`.

Figure 6.8 shows the call stack and heap *during* and *after* executing `createArray`. See Review Question 6.14 to show the call stack and heap for other methods in the program.

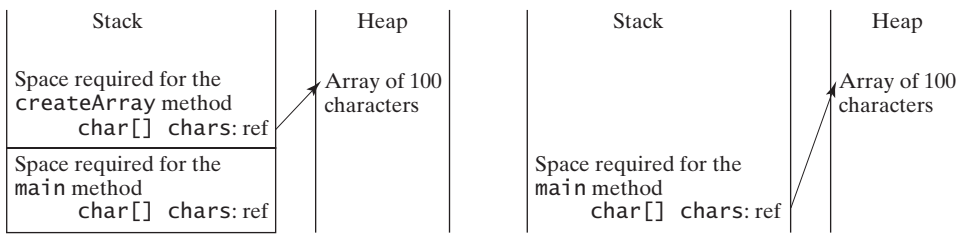


FIGURE 6.8 (a) An array of 100 characters is created when executing `createArray`. (b) This array is returned and assigned to the variable `chars` in the `main` method.

6.8 Variable-Length Argument Lists

You can pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:

```
typeName... parameterName
```

In the method declaration, you specify the type followed by an ellipsis (. . .). Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Java treats a variable-length parameter as an array. You can pass an array or a variable number of arguments to a variable-length parameter. When invoking a method with a variable number of arguments, Java creates an array and passes the arguments to it. Listing 6.5 contains a method that prints the maximum value in a list of an unspecified number of values.

LISTING 6.5 VarArgsDemo.java

```
1 public class VarArgsDemo {
2   public static void main(String[] args) {
3     printMax(34, 3, 3, 2, 56.5);
```

pass variable-length arg list

pass an array arg

```
4     printMax(new double[]{1, 2, 3});
5   }
6
7   public static void printMax(double... numbers) {
8     if (numbers.length == 0) {
9       System.out.println("No argument passed");
10      return;
11    }
12
13    double result = numbers[0];
14
15    for (int i = 1; i < numbers.length; i++)
16      if (numbers[i] > result)
17        result = numbers[i];
18
19    System.out.println("The max value is " + result);
20  }
21 }
```

a variable-length arg parameter

Line 3 invokes the `printMax` method with a variable-length argument list passed to the array `numbers`. If no arguments are passed, the length of the array is `0` (line 8).

Line 4 invokes the `printMax` method with an array.

6.9 Searching Arrays

Searching is the process of looking for a specific element in an array—for example, discovering whether a certain score is included in a list of scores. Searching is a common task in computer programming. Many algorithms and data structures are devoted to searching. This section discusses two commonly used approaches, *linear search* and *binary search*.

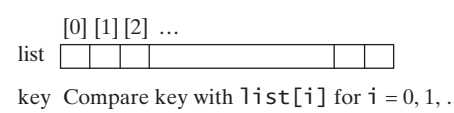
linear search
binary search

6.9.1 The Linear Search Approach

The linear search approach compares the key element `key` sequentially with each element in the array. It continues to do so until the key matches an element in the array or the array is exhausted without a match being found. If a match is made, the linear search returns the index of the element in the array that matches the key. If no match is found, the search returns `-1`. The `linearSearch` method in Listing 6.6 gives the solution:

LISTING 6.6 LinearSearch.java

```
1 public class LinearSearch {
2   /** The method for finding a key in the list */
3   public static int linearSearch(int[] list, int key) {
4     for (int i = 0; i < list.length; i++) {
5       if (key == list[i])
6         return i;
7     }
8     return -1;
9   }
10 }
```



To better understand this method, trace it with the following statements:

```
int[] list = {1, 4, 4, 2, 5, -3, 6, 2};
int i = linearSearch(list, 4);  // Returns 1
int j = linearSearch(list, -4); // Returns -1
int k = linearSearch(list, -3); // Returns 5
```

The linear search method compares the key with each element in the array. The elements can be in any order. On average, the algorithm will have to compare half of the elements in an

array before finding the key, if it exists. Since the execution time of a linear search increases linearly as the number of array elements increases, linear search is inefficient for a large array.

6.9.2 The Binary Search Approach

Binary search is the other common search approach for a list of values. For binary search to work, the elements in the array must already be ordered. Assume that the array is in ascending order. The binary search first compares the key with the element in the middle of the array. Consider the following three cases:

- If the key is less than the middle element, you need to continue to search for the key only in the first half of the array.
- If the key is equal to the middle element, the search ends with a match.
- If the key is greater than the middle element, you need to continue to search for the key only in the second half of the array.

Clearly, the binary search method eliminates half of the array after each comparison. Sometimes you eliminate half of the elements, and sometimes you eliminate half plus one. Suppose that the array has *n* elements. For convenience, let `n` be a power of `2`. After the first comparison, `n/2` elements are left for further search; after the second comparison, `(n/2)/2` elements are left. After the `k`th comparison, $n/2^k$ elements are left for further search. When $k = \log_2 n$, only one element is left in the array, and you need only one more comparison. Therefore, in the worst case when using the binary search approach, you need $\log_2 n + 1$ comparisons to find an element in the sorted array. In the worst case for a list of `1024` (2^{10}) elements, binary search requires only `11` comparisons, whereas a linear search requires `1023` comparisons in the worst case.

The portion of the array being searched shrinks by half after each comparison. Let `low` and `high` denote, respectively, the first index and last index of the array that is currently being searched. Initially, `low` is `0` and `high` is `list.length-1`. Let `mid` denote the index of the middle element. So `mid` is `(low + high)/2`. Figure 6.9 shows how to find key `11` in the list {`2`, `4`, `7`, `10`, `11`, `45`, `50`, `59`, `60`, `66`, `69`, `70`, `79`} using binary search.

You now know how the binary search works. The next task is to implement it in Java. Don't rush to give a complete implementation. Implement it incrementally, one step at a time. You may start with the first iteration of the search, as shown in Figure 6.10(a). It compares the key with the middle element in the list whose `low` index is `0` and `high` index is `list.length - 1`.

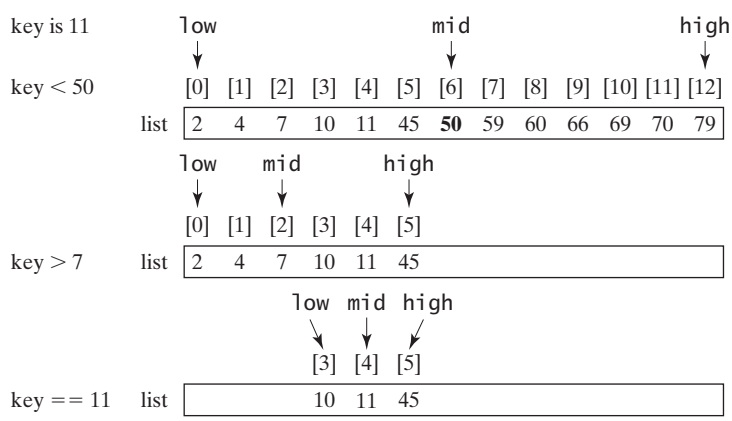


FIGURE 6.9 Binary search eliminates half of the list from further consideration after each comparison.

If `key < list[mid]`, set the `high` index to `mid - 1`; if `key == list[mid]`, a match is found and return `mid`; if `key > list[mid]`, set the `low` index to `mid + 1`.

```
public static int binarySearch(
    int[] list, int key) {
  int low = 0;
  int high = list.length - 1;


    int mid = (low + high) / 2;
    if (key < list[mid])
      high = mid - 1;
    else if (key == list[mid])
      return mid;
    else
      low = mid + 1;


}
```

(a) Version 1

```
public static int binarySearch(
    int[] list, int key) {
  int low = 0;
  int high = list.length - 1;

  while (high >= low) {
    int mid = (low + high) / 2;
    if (key < list[mid])
      high = mid - 1;
    else if (key == list[mid])
      return mid;
    else
      low = mid + 1;
  }

  return -1; // Not found
}
```

(b) Version 2

FIGURE 6.10 Binary search is implemented incrementally.

Next consider implementing the method to perform search repeatedly by adding a loop, as shown in Figure 6.10(b). The search ends if the key is found, or if the key is not found when `low > high`.

why not `-1`?

When the key is not found, `low` is the insertion point where a key would be inserted to maintain the order of the list. It is more useful to return the insertion point than `-1`. The method must return a negative value to indicate that the key is not in the list. Can it simply return `-low`? No. If key is less than `list[0]`, `low` would be `0`. `-0` is `0`. This would indicate that key matches `list[0]`. A good choice is to let the method return `-low - 1` if the key is not in the list. Returning `-low - 1` indicates not only that the key is not in the list, but also where the key would be inserted.

The complete program is given in Listing 6.7.

LISTING 6.7 BinarySearch.java

```
1 public class BinarySearch {
2   /** Use binary search to find the key in the list */
3   public static int binarySearch(int[] list, int key) {
4     int low = 0;
5     int high = list.length - 1;
6
7     while (high >= low) {
8       int mid = (low + high) / 2;
9       if (key < list[mid])
10        high = mid - 1;
11      else if (key == list[mid])
12        return mid;
13      else
14        low = mid + 1;
15    }
16
17    return -low - 1; // Now high < low, key not found
18  }
19 }
```

first half

second half

The binary search returns the index of the search key if it is contained in the list (line 12). Otherwise, it returns `-low - 1` (line 17).

What would happens if we replaced `(high >= low)` in line 7 with `(high > low)`? The search would miss a possible matching element. Consider a list with just one element. The search would miss the element.

Does the method still work if there are duplicate elements in the list? Yes, as long as the elements are sorted in increasing order. The method returns the index of one of the matching elements if the element is in the list.

To better understand this method, trace it with the following statements and identify `low` and `high` when the method returns.

```
int[] list = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
int i = BinarySearch.binarySearch(list, 2); // Returns 0
int j = BinarySearch.binarySearch(list, 11); // Returns 4
int k = BinarySearch.binarySearch(list, 12); // Returns -6
int l = BinarySearch.binarySearch(list, 1); // Returns -1
int m = BinarySearch.binarySearch(list, 3); // Returns -2
```

Here is the table that lists the `low` and `high` values when the method exits and the value returned from invoking the method.

Method	Low	High	Value Returned
binarySearch(list, 2)	0	1	0
binarySearch(list, 11)	3	5	4
binarySearch(list, 12)	5	4	−6
binarySearch(list, 1)	0	−1	−1
binarySearch(list, 3)	1	0	−2



Note

Linear search is useful for finding an element in a small array or an unsorted array, but it is inefficient for large arrays. Binary search is more efficient, but it requires that the array be presorted.

binary search benefits

6.10 Sorting Arrays

Sorting, like searching, is a common task in computer programming. Many different algorithms have been developed for sorting. This section introduces two simple, intuitive sorting algorithms: *selection sort* and *insertion sort*.

6.10.1 Selection Sort



Video Note
Selection sort

Suppose that you want to sort a list in ascending order. Selection sort finds the smallest number in the list and places it first. It then finds the smallest number remaining and places it next to first, and so on, until only a single number remains. Figure 6.11 shows how to sort a list {2, 9, 5, 4, 8, 1, 6} using selection sort.

You know how the selection-sort approach works. The task now is to implement it in Java. Beginners find it difficult to develop a complete solution on the first attempt. Start by writing the code for the first iteration to find the largest element in the list and swap it with the last element, and then observe what would be different for the second iteration, the third, and so on. The insight this gives will enable you to write a loop that generalizes all the iterations.

The solution can be described as follows:

```
for (int i = 0; i < list.length - 1; i++) {
  select the smallest element in list[i..list.length-1];
  swap the smallest with list[i], if necessary;
    // list[i] is in its correct position.
    // The next iteration apply on list[i+1..list.length-1]
}
```

Listing 6.8 implements the solution.

LISTING 6.8 SelectionSort.java

```
1  public class SelectionSort {
2    /** The method for sorting the numbers */
3    public static void selectionSort(double[] list) {
4      for (int i = 0; i < list.length - 1; i++) {
5        // Find the minimum in the list[i..list.length-1]
6        double currentMin = list[i];
7        int currentMinIndex = i;
8
9        for (int j = i + 1; j < list.length; j++) {
10         if (currentMin > list[j]) {
11           currentMin = list[j];
12           currentMinIndex = j;
13         }
14       }
15
16       // Swap list[i] with list[currentMinIndex] if necessary;
17       if (currentMinIndex != i) {
18         list[currentMinIndex] = list[i];
19         list[i] = currentMin;
20       }
21     }
22   }
23 }
```

select

swap

The `selectionSort(double[] list)` method sorts any array of `double` elements. The method is implemented with a nested `for` loop. The outer loop (with the loop control variable `i`) (line 4) is iterated in order to find the smallest element in the list, which ranges from `list[i]` to `list[list.length-1]`, and exchange it with `list[i]`.

The variable `i` is initially `0`. After each iteration of the outer loop, `list[i]` is in the right place. Eventually, all the elements are put in the right place; therefore, the whole list is sorted.

To understand this method better, trace it with the following statements:

```
double[] list = {1, 9, 4.5, 6.6, 5.7, -4.5};
SelectionSort.selectionSort(list);
```

6.10.2 Insertion Sort

Suppose that you want to sort a list in ascending order. The insertion-sort algorithm sorts a list of values by repeatedly inserting a new element into a sorted sublist until the whole list is sorted. Figure 6.12 shows how to sort the list {2, 9, 5, 4, 8, 1, 6} using insertion sort.

The algorithm can be described as follows:

```
for (int i = 1; i < list.length; i++) {
  insert list[i] into a sorted sublist list[0..i-1] so that
  list[0..i] is sorted.
}
```

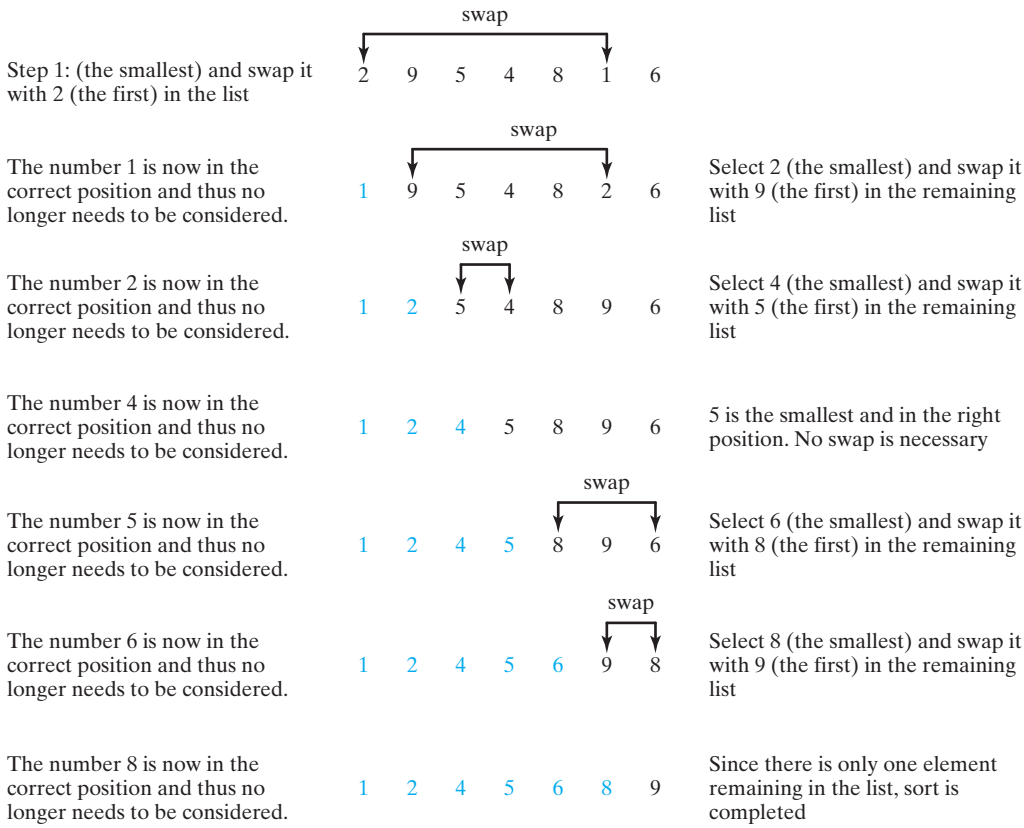

swap
Step 1: (the smallest) and swap it with 2 (the first) in the list
2 9 5 4 8 1 6
swap
The number 1 is now in the correct position and thus no longer needs to be considered.
1 9 5 4 8 2 6
Select 2 (the smallest) and swap it with 9 (the first) in the remaining list
swap
The number 2 is now in the correct position and thus no longer needs to be considered.
1 2 5 4 8 9 6
Select 4 (the smallest) and swap it with 5 (the first) in the remaining list
The number 4 is now in the correct position and thus no longer needs to be considered.
1 2 4 5 8 9 6
5 is the smallest and in the right position. No swap is necessary
swap
The number 5 is now in the correct position and thus no longer needs to be considered.
1 2 4 5 8 9 6
Select 6 (the smallest) and swap it with 8 (the first) in the remaining list
swap
The number 6 is now in the correct position and thus no longer needs to be considered.
1 2 4 5 6 9 8
Select 8 (the smallest) and swap it with 9 (the first) in the remaining list
The number 8 is now in the correct position and thus no longer needs to be considered.
1 2 4 5 6 8 9
Since there is only one element remaining in the list, sort is completed

FIGURE 6.11 Selection sort repeatedly selects the smallest number and swaps it with the first number in the list.

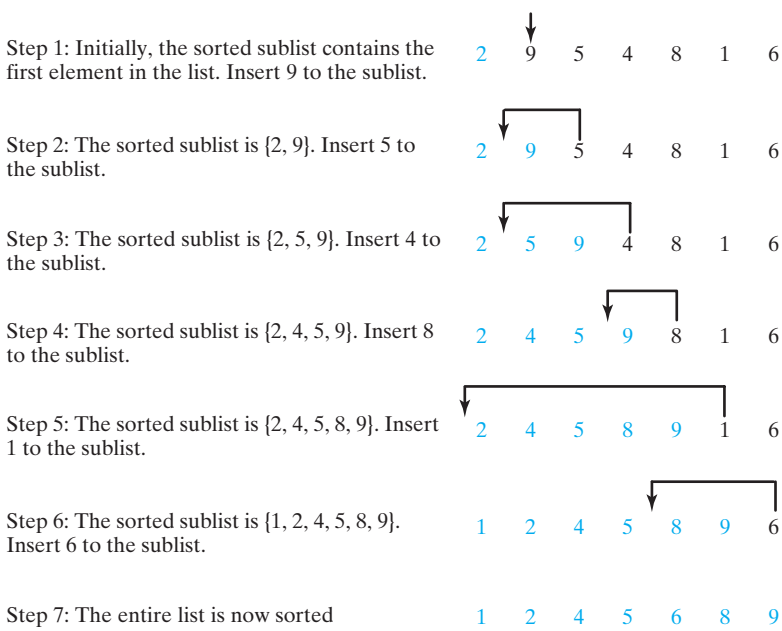

Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 to the sublist.
2 9 5 4 8 1 6
Step 2: The sorted sublist is {2, 9}. Insert 5 to the sublist.
2 9 5 4 8 1 6
Step 3: The sorted sublist is {2, 5, 9}. Insert 4 to the sublist.
2 5 9 4 8 1 6
Step 4: The sorted sublist is {2, 4, 5, 9}. Insert 8 to the sublist.
2 4 5 9 8 1 6
Step 5: The sorted sublist is {2, 4, 5, 8, 9}. Insert 1 to the sublist.
2 4 5 8 9 1 6
Step 6: The sorted sublist is {1, 2, 4, 5, 8, 9}. Insert 6 to the sublist.
1 2 4 5 8 9 6
Step 7: The entire list is now sorted
1 2 4 5 6 8 9

FIGURE 6.12 Insertion sort repeatedly inserts a new element into a sorted sublist.

To insert `list[i]` into `list[0..i-1]`, save `list[i]` into a temporary variable, say `currentElement`. Move `list[i-1]` to `list[i]` if `list[i-1] > currentElement`, move `list[i-2]` to `list[i-1]` if `list[i-2] > currentElement`, and so on, until `list[i-k] <= currentElement` or `k > i` (we pass the first element of the sorted list). Assign `currentElement` to `list[i-k+1]`. For example, to insert `4` into {`2`, `5`, `9`} in Step 3 in Figure 6.13, move `list[2]` (`9`) to `list[3]` since `9 > 4`, move `list[1]` (`5`) to `list[2]` since `5 > 4`. Finally, move `currentElement` (`4`) to `list[1]`.

The algorithm can be expanded and implemented as in Listing 6.9.

LISTING 6.9 InsertionSort.java

```
1  public class InsertionSort {
2    /** The method for sorting the numbers */
3    public static void insertionSort(double[] list) {
4      for (int i = 1; i < list.length; i++) {
5        /** insert list[i] into a sorted sublist list[0..i-1] so that
6            list[0..i] is sorted. */
7        double currentElement = list[i];
8        int k;
9        for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {
10         list[k + 1] = list[k];
11       }
12
13       // Insert the current element into list[k + 1]
14       list[k + 1] = currentElement;
15     }
16   }
17 }
```

shift

insert

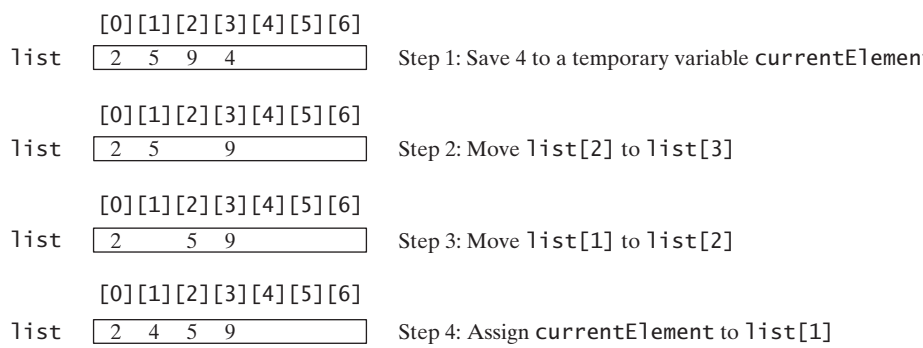


FIGURE 6.13 A new element is inserted into a sorted sublist.

The `insertionSort(double[] list)` method sorts any array of `double` elements. The method is implemented with a nested `for` loop. The outer loop (with the loop control variable `i`) (line 4) is iterated in order to obtain a sorted sublist, which ranges from `list[0]` to `list[i]`. The inner loop (with the loop control variable `k`) inserts `list[i]` into the sublist from `list[0]` to `list[i-1]`.

To better understand this method, trace it with the following statements:

```
double[] list = {1, 9, 4.5, 6.6, 5.7, -4.5};
InsertionSort.insertionSort(list);
```

6.11 The **Arrays** Class

The **java.util.Arrays** class contains various static methods for sorting and searching arrays, comparing arrays, and filling array elements. These methods are overloaded for all primitive types.

sort

You can use the **sort** method to sort a whole array or a partial array. For example, the following code sorts an array of numbers and an array of characters.

```
double[] numbers = {6.0, 4.4, 1.9, 2.9, 3.4, 3.5};
java.util.Arrays.sort(numbers); // Sort the whole array

char[] chars = {'a', 'A', '4', 'F', 'D', 'P'};
java.util.Arrays.sort(chars, 1, 3); // Sort part of the array
```

Invoking **sort(numbers)** sorts the whole array **numbers**. Invoking **sort(chars, 1, 3)** sorts a partial array from **chars[1]** to **chars[3-1]**.

binarySearch

You can use the **binarySearch** method to search for a key in an array. The array must be presorted in increasing order. If the key is not in the array, the method returns ***-(insertion index + 1)***. For example, the following code searches the keys in an array of integers and an array of characters.

```
int[] list = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
System.out.println("(1) Index is " +
  java.util.Arrays.binarySearch(list, 11));
System.out.println("(2) Index is " +
  java.util.Arrays.binarySearch(list, 12));

char[] chars = {'a', 'c', 'g', 'x', 'y', 'z'};
System.out.println("(3) Index is " +
  java.util.Arrays.binarySearch(chars, 'a'));
System.out.println("(4) Index is " +
  java.util.Arrays.binarySearch(chars, 't'));
```

The output of the preceding code is

```
(1) Index is 4
(2) Index is -6
(3) Index is 0
(4) Index is -4
```

equals

You can use the **equals** method to check whether two arrays are equal. Two arrays are equal if they have the same contents. In the following code, **list1** and **list2** are equal, but **list2** and **list3** are not.

```
int[] list1 = {2, 4, 7, 10};
int[] list2 = {2, 4, 7, 10};
int[] list3 = {4, 2, 7, 10};
System.out.println(java.util.Arrays.equals(list1, list2)); // true
System.out.println(java.util.Arrays.equals(list2, list3)); // false
```

fill

You can use the **fill** method to fill in all or part of the array. For example, the following code fills **list1** with **5** and fills **8** into elements **list2[1]** and **list2[3-1]**.

```
int[] list1 = {2, 4, 7, 10};
int[] list2 = {2, 4, 7, 10};
java.util.Arrays.fill(list1, 5); // Fill 5 to the whole array
java.util.Arrays.fill(list2, 1, 3, 8); // Fill 8 to a partial array
```

Key Terms

anonymous array 209
array 198
array initializer 201
binary search 216
garbage collection 208
index 198
indexed variable 200
insertion sort 219
linear search 216
selection sort 219

Chapter Summary

1. A variable is declared as an array type using the syntax `elementType[] arrayRefVar` or `elementType arrayRefVar[]`. The style `elementType[] arrayRefVar` is preferred, although `elementType arrayRefVar[]` is legal.

2. Unlike declarations for primitive data type variables, the declaration of an array variable does not allocate any space in memory for the array. An array variable is not a primitive data type variable. An array variable contains a reference to an array.

3. You cannot assign elements to an array unless it has already been created. You can create an array by using the `new` operator with the following syntax: `new elementType[arraySize]`.

4. Each element in the array is represented using the syntax `arrayRefVar[index]`. An index must be an integer or an integer expression.

5. After an array is created, its size becomes permanent and can be obtained using `arrayRefVar.length`. Since the index of an array always begins with `0`, the last index is always `arrayRefVar.length - 1`. An out-of-bounds error will occur if you attempt to reference elements beyond the bounds of an array.

6. Programmers often mistakenly reference the first element in an array with index `1`, but it should be `0`. This is called the *index off-by-one error*.

7. When an array is created, its elements are assigned the default value of `0` for the numeric primitive data types, `'\u0000'` for char types, and `false` for `boolean` types.

8. Java has a shorthand notation, known as the *array initializer*, which combines in one statement declaring an array, creating an array, and initializing, using the syntax: `elementType[] arrayRefVar = {value0, value1, ..., valuek}`.

9. When you pass an array argument to a method, you are actually passing the reference of the array; that is, the called method can modify the elements in the caller's original array.

Review Questions

Section 6.2

6.1 How do you declare and create an array?

6.2 How do you access elements of an array?

6.3 Is memory allocated for an array when it is declared? When is the memory allocated for an array? What is the printout of the following code?

```
int x = 30;
int[] numbers = new int[x];
x = 60;
System.out.println("x is " + x);
System.out.println("The size of numbers is " + numbers.length);
```

6.4 Indicate true or false for the following statements:

- Every element in an array has the same type.
- The array size is fixed after it is declared.
- The array size is fixed after it is created.
- The elements in an array must be of primitive data type.

6.5 Which of the following statements are valid array declarations?

```
int i = new int(30);
double d[] = new double[30];
char[] r = new char(1..30);
int i[] = (3, 4, 3, 2);
float f[] = {2.3, 4.5, 6.6};
char[] c = new char();
```

6.6 What is the array index type? What is the lowest index? What is the representation of the third element in an array named `a`?

6.7 Write statements to do the following:

a. Create an array to hold `10` double values.
b. Assign value `5.5` to the last element in the array.
c. Display the sum of the first two elements.
d. Write a loop that computes the sum of all elements in the array.
e. Write a loop that finds the minimum element in the array.
f. Randomly generate an index and display the element of this index in the array.
g. Use an array initializer to create another array with initial values `3.5`, `5.5`, `4.52`, and `5.6`.

6.8 What happens when your program attempts to access an array element with an invalid index?

6.9 Identify and fix the errors in the following code:

```
1 public class Test {
2   public static void main(String[] args) {
3     double[100] r;
4
5     for (int i = 0; i < r.length(); i++);
6       r(i) = Math.random * 100;
7   }
8 }
```

Section 6.3

6.10 Use the `arraycopy()` method to copy the following array to a target array `t`:

```
int[] source = {3, 4, 5};
```

6.11 Once an array is created, its size cannot be changed. Does the following code resize the array?

```
int[] myList;
myList = new int[10];
// Some time later you want to assign a new array to myList
myList = new int[20];
```

Sections 6.4–6.7

6.12 When an array is passed to a method, a new array is created and passed to the method. Is this true?

6.13 Show the output of the following two programs:

```
public class Test {
  public static void main(String[] args) {
    int number = 0;
    int[] numbers = new int[1];

    m(number, numbers);

    System.out.println("number is " + number
      + " and numbers[0] is " + numbers[0]);
  }

  public static void m(int x, int[] y) {
    x = 3;
    y[0] = 3;
  }
}
```

a

```
public class Test {
  public static void main(String[] args) {
    int[] list = {1, 2, 3, 4, 5};
    reverse(list);
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
  }

  public static void  reverse(int[] list) {
    int[] newList = new int[list.length];

    for (int i = 0; i < list.length; i++)
      newList[i] = list[list.length - 1 - i];

    list = newList;
  }
}
```

b

6.14 Where are the arrays stored during execution? Show the contents of the stack and heap during and after executing `createArray`, `displayArray`, `countLetters`, `displayCounts` in Listing 6.4.

Section 6.8

6.15 What is wrong in the following method declaration?

```
public static void print(String... strings, double... numbers)
public static void print(double... numbers, String name)
public static double... print(double d1, double d2)
```

6.16 Can you invoke the `printMax` method in Listing 6.5 using the following statements?

```
printMax(1, 2, 2, 1, 4);
printMax(new double[]{1, 2, 3});
printMax(new int[]{1, 2, 3});
```

Sections 6.9–6.10

6.17 Use Figure 6.9 as an example to show how to apply the binary search approach to a search for key 10 and key 12 in list {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79}.

6.18 Use Figure 6.11 as an example to show how to apply the selection-sort approach to sort {3.4, 5, 3, 3.5, 2.2, 1.9, 2}.

6.19 Use Figure 6.12 as an example to show how to apply the insertion-sort approach to sort {3.4, 5, 3, 3.5, 2.2, 1.9, 2}.

6.20 How do you modify the `selectionSort` method in Listing 6.8 to sort numbers in decreasing order?

6.21 How do you modify the `insertionSort` method in Listing 6.9 to sort numbers in decreasing order?

Section 6.11

6.22 What types of array can be sorted using the `java.util.Arrays.sort` method? Does this `sort` method create a new array?

6.23 To apply `java.util.Arrays.binarySearch(array, key)`, should the array be sorted in increasing order, in decreasing order, or neither?

6.24 Show the contents of the array after the execution of each line.

```
int[] list = {2, 4, 7, 10};
java.util.Arrays.fill(list, 7);
java.util.Arrays.fill(list, 1, 3, 8);
System.out.print(java.util.Arrays.equals(list, list));
```

Programming Exercises

Section 6.2

6.1* (*Assigning grades*) Write a program that reads student scores, gets the best score, and then assigns grades based on the following scheme:

Grade is A if score is >= best − 10;
Grade is B if score is >= best − 20;
Grade is C if score is >= best − 30;
Grade is D if score is >= best − 40;
Grade is F otherwise.

The program prompts the user to enter the total number of students, then prompts the user to enter all of the scores, and concludes by displaying the grades. Here is a sample run:

```
Enter the number of students: 4 Enter
Enter 4 scores: 40 55 70 58 Enter
Student 0 score is 40 and grade is C
Student 1 score is 55 and grade is B
Student 2 score is 70 and grade is A
Student 3 score is 58 and grade is B
```



6.2 (*Reversing the numbers entered*) Write a program that reads ten integers and displays them in the reverse of the order in which they were read.

6.3** (*Counting occurrence of numbers*) Write a program that reads the integers between 1 and 100 and counts the occurrences of each. Assume the input ends with 0. Here is a sample run of the program:



```
Enter the integers between 1 and 100: 2 5 6 5 4 3 23
  43 2 0 Enter
2 occurs 2 times
3 occurs 1 time
4 occurs 1 time
5 occurs 2 times
6 occurs 1 time
23 occurs 1 time
43 occurs 1 time
```

Note that if a number occurs more than one time, the plural word "times" is used in the output.

6.4 (*Analyzing scores*) Write a program that reads an unspecified number of scores and determines how many scores are above or equal to the average and how many scores are below the average. Enter a negative number to signify the end of the input. Assume that the maximum number of scores is `10`.

6.5** (*Printing distinct numbers*) Write a program that reads in ten numbers and displays distinct numbers (i.e., if a number appears multiple times, it is displayed only once). *Hint*: Read a number and store it to an array if it is new. If the number is already in the array, ignore it. After the input, the array contains the distinct numbers. Here is the sample run of the program:



```
Enter ten numbers: 1 2 3 2 1 6 3 4 5 2 Enter
The distinct numbers are: 1 2 3 6 4 5
```

6.6* (*Revising Listing 4.14, PrimeNumber.java*) Listing 4.14 determines whether a number `n` is prime by checking whether `2`, `3`, `4`, `5`, `6`, ..., `n/2` is a divisor. If a divisor is found, `n` is not prime. A more efficient approach is to check whether any of the prime numbers less than or equal to $\sqrt{n}$ can divide `n` evenly. If not, `n` is prime. Rewrite Listing 4.11 to display the first 50 prime numbers using this approach. You need to use an array to store the prime numbers and later use them to check whether they are possible divisors for `n`.

6.7* (*Counting single digits*) Write a program that generates `100` random integers between `0` and `9` and displays the count for each number. (*Hint*: Use `(int)(Math.random() * 10)` to generate a random integer between 0 and 9. Use an array of ten integers, say `counts`, to store the counts for the number of 0s, 1s, ..., 9s.)

Sections 6.4–6.7

6.8 (*Averaging an array*) Write two overloaded methods that return the average of an array with the following headers:

```
public static int average(int[] array)
public static double average(double[] array)
```

Write a test program that prompts the user to enter ten double values, invokes this method, and displays the average value.

6.9 (*Finding the smallest element*) Write a method that finds the smallest element in an array of integers using the following header:

```
public static double min(double[] array)
```

Write a test program that prompts the user to enter ten numbers, invokes this method to return the minimum value, and displays the minimum value. Here is a sample run of the program:

```
Enter ten numbers: 1.9 2.5 3.7 2 1.5 6 3 4 5 2 ↵Enter
The minimum number is: 1.5
```



6.10 (*Finding the index of the smallest element*) Write a method that returns the index of the smallest element in an array of integers. If the number of such elements is greater than 1, return the smallest index. Use the following header:

```
public static int indexOfSmallestElement(double[] array)
```

Write a test program that prompts the user to enter ten numbers, invokes this method to return the index of the smallest element, and displays the index.

6.11* (*Statistics: computing deviation*) Exercise 5.21 computes the standard deviation of numbers. This exercise uses a different but equivalent formula to compute the standard deviation of **n** numbers.

$$mean = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{x_1 + x_2 + \ldots + x_n}{n} \qquad deviation = \sqrt{\frac{\sum_{i=1}^{n} (x_i - mean)^2}{n - 1}}$$

To compute deviation with this formula, you have to store the individual numbers using an array, so that they can be used after the mean is obtained.

Your program should contain the following methods:

```
/** Compute the deviation of double values*/
public static double deviation(double[] x)

/** Compute the mean of an array of double values*/
public static double mean(double[] x)
```

Write a test program that prompts the user to enter ten numbers and displays the mean and deviation, as shown in the following sample run:

```
Enter ten numbers: 1.9 2.5 3.7 2 1 6 3 4 5 2 ↵Enter
The mean is 3.11
The standard deviation is 1.55738
```



6.12* (*Reversing an array*) The **reverse** method in §6.7 reverses an array by copying it to a new array. Rewrite the method that reverses the array passed in the argument and returns this array. Write a test program that prompts the user to enter ten numbers, invokes the method to reverse the numbers, and displays the numbers.

Section 6.8

6.13* (*Random number chooser*) Write a method that returns a random number between **1** and **54**, excluding the numbers passed in the argument. The method header is specified as follows:

```
public static int getRandom(int... numbers)
```

6.14 (*Computing gcd*) Write a method that returns the gcd of an unspecified number of integers. The method header is specified as follows:

```
public static int gcd(int... numbers)
```

Write a test program that prompts the user to enter five numbers, invokes the method to find the gcd of these numbers, and displays the gcd.

Sections 6.9–6.10

6.15 (*Eliminating duplicates*) Write a method to eliminate the duplicate values in the array using following method header:

```
public static int[] eliminateDuplicates(int[] numbers)
```

Write a test program that reads in ten integers, invokes the method, and displays the result. Here is the sample run of the program:



```
Enter ten numbers: 1 2 3 2 1 6 3 4 5 2 Enter
The distinct numbers are: 1 2 3 6 4 5
```

6.16 (*Execution time*) Write a program that randomly generates an array of **100000** integers and a key. Estimate the execution time of invoking the **linearSearch** method in Listing 6.6. Sort the array and estimate the execution time of invoking the **binarySearch** method in Listing 6.7. You can use the following code template to obtain the execution time:

```
long startTime = System.currentTimeMillis();
perform the task;
long endTime = System.currentTimeMillis();
long executionTime = endTime - startTime;
```

6.17* (*Revising selection sort*) In §6.10.1, you used selection sort to sort an array. The selection-sort method repeatedly finds the smallest number in the current array and swaps it with the first number in the array. Rewrite this program by finding the largest number and swapping it with the last number in the array. Write a test program that reads in ten double numbers, invokes the method, and displays the sorted numbers.

6.18** (*Bubble sort*) Write a sort method that uses the bubble-sort algorithm. The bubble-sort algorithm makes several passes through the array. On each pass, successive neighboring pairs are compared. If a pair is in decreasing order, its values are swapped; otherwise, the values remain unchanged. The technique is called a *bubble sort* or *sinking sort* because the smaller values gradually "bubble" their way to the top and the larger values "sink" to the bottom. Use {**6.0**, **4.4**, **1.9**, **2.9**, **3.4**, **2.9**, **3.5**} to test the method. Write a test program that reads in ten double numbers, invokes the method, and displays the sorted numbers.

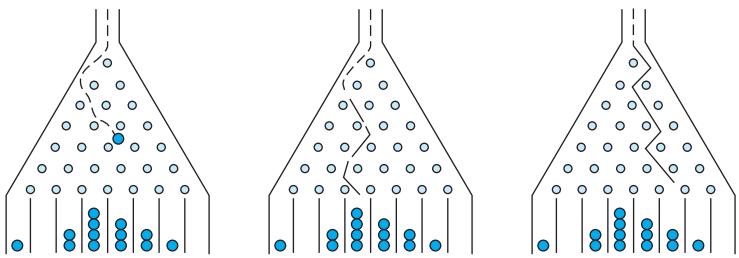
6.19** (*Sorting students*) Write a program that prompts the user to enter the number of students, the students' names, and their scores, and prints student names in decreasing order of their scores.

6.20*** (*Game: Eight Queens*) The classic Eight Queens puzzle is to place eight queens on a chessboard such that no two queens can attack each other (i.e., no two queens are on the same row, same column, or same diagonal). There are many possible solutions. Write a program that displays one such solution. A sample

output is shown below:

```
|Q| | | | | | | |
| | | | |Q| | | |
| | | | | | | |Q|
| | | | | |Q| | |
| | |Q| | | | | |
| | | | | | |Q| |
| |Q| | | | | | |
| | | |Q| | | | |
```

6.21* (*Game: bean machine*) The bean machine, also known as a quincunx or the Galton box, is a device for statistic experiments named after English scientist Sir Francis Galton. It consists of an upright board with evenly spaced nails (or pegs) in a triangular form, as shown in Figure 6.14.



(a) (b) (c)

FIGURE 6.14 Each ball takes a random path and falls into a slot.

Balls are dropped from the opening of the board. Every time a ball hits a nail, it has a 50% chance of falling to the left or to the right. The piles of balls are accumulated in the slots at the bottom of the board.

Write a program that simulates the bean machine. Your program should prompt the user to enter the number of the balls and the number of the slots in the machine. Simulate the falling of each ball by printing its path. For example, the path for the ball in Figure 6.14(b) is LLRRLLR and the path for the ball in Figure 6.14(c) is RLRRLRR. Display the final buildup of the balls in the slots in a histogram. Here is a sample run of the program:



```
Enter the number of balls to drop: 5 Enter
Enter the number of slots in the bean machine: 7 Enter

LRLRLRR
RRLLLRR
LLRLLRR
RRLLLLL
LRLRRLR

  0
  0
000
```

(*Hint*: Create an array named `slots`. Each element in `slots` stores the number of balls in a slot. Each ball falls into a slot via a path. The number of R's in a path is the position of the slot where the ball falls. For example, for the path

LRLRLRR, the ball falls into `slots[4]`, and for the path is RRLLLLL, the ball falls into `slots[2]`.)

6.22* (*Game: multiple Eight Queens solutions)* Exercise 6.20 finds one solution for the Eight Queens problem. Write a program to count all possible solutions for the eight queens problem and display all solutions.

6.23** (*Game: locker puzzle*) A school has 100 lockers and 100 students. All lockers are closed on the first day of school. As the students enter, the first student, denoted S1, opens every locker. Then the second student, S2, begins with the second locker, denoted L2, and closes every other locker. Student S3 begins with the third locker and changes every third locker (closes it if it was open, and opens it if it was closed). Student S4 begins with locker L4 and changes every fourth locker. Student S5 starts with L5 and changes every fifth locker, and so on, until student S100 changes L100.

After all the students have passed through the building and changed the lockers, which lockers are open? Write a program to find your answer.

(*Hint*: Use an array of `100 boolean` elements, each of which indicates whether a locker is open (`true`) or closed (`false`). Initially, all lockers are closed.)

6.24** (*Simulation: coupon collector's problem*) Coupon collector is a classic statistic problem with many practical applications. The problem is to pick objects from a set of objects repeatedly and find out how many picks are needed for all the objects to be picked at least once. A variation of the problem is to pick cards from a shuffled deck of `52` cards repeatedly and find out how many picks are needed before you see one of each suit. Assume a picked card is placed back in the deck before picking another. Write a program to simulate the number of picks needed to get four cards from each suit and display the four cards picked (it is possible a card may be picked twice). Here is a sample run of the program:



Video Note
Coupon collector's problem



```
Queen of Spades
5 of Clubs
Queen of Hearts
4 of Diamonds
Number of picks: 12
```

6.25 (*Algebra: solving quadratic equations*) Write a method for solving a quadratic equation using the following header:

```
public static int solveQuadratic(double[] eqn, double[] roots)
```

The coefficients of a quadratic equation $ax^2 + bx + c = 0$ are passed to the array `eqn` and the noncomplex roots are stored in roots. The method returns the number of roots. See Programming Exercise 3.1 on how to solve a quadratic equation.

Write a program that prompts the user to enter values for a, b, and c and displays the number of roots and all noncomplex roots.

6.26 (*Strictly identical arrays*) Two arrays `list1` and `list2` are *strictly identical* if they have the same length and `list1[i]` is equal to `list2[i]` for each `i`. Write a method that returns `true` if `list1` and `list2` are strictly identical, using the following header:

```
public static boolean equal(int[] list1, int[] list2)
```

Write a test program that prompts the user to enter two lists of integers and displays whether the two are strictly identical. Here are the sample runs. Note that the first number in the input indicates the number of the elements in the list.

```
Enter list1: 5 2 5 6 1 6 Enter
Enter list2: 5 2 5 6 1 6 Enter
Two lists are strictly identical
```



```
Enter list1: 5 2 5 6 6 1 Enter
Enter list2: 5 2 5 6 1 6 Enter
Two lists are not strictly identical
```



6.27 (*Identical arrays*) Two arrays `list1` and `list2` are *identical* if they have the same contents. Write a method that returns `true` if `list1` and `list2` are identical, using the following header:

```
public static boolean equal(int[] list1, int[] list2)
```

Write a test program that prompts the user to enter two lists of integers and displays whether the two are identical. Here are the sample runs. Note that the first number in the input indicates the number of the elements in the list.

```
Enter list1: 5 2 5 6 6 1 Enter
Enter list2: 5 5 2 6 1 6 Enter
Two lists are identical
```



```
Enter list1: 5 5 5 6 6 1 Enter
Enter list2: 5 2 5 6 1 6 Enter
Two lists are not identical
```



6.28 (*Math: combinations*) Write a program that prompts the user to enter 10 integers and displays all combinations of picking two numbers from the 10.

6.29 (*Game: picking four cards*) Write a program that picks four cards from a deck of 52 cards and computes their sum. An Ace, King, Queen, and Jack represent 1, 13, 12, and 11, respectively. Your program should display the number of picks that yields the sum of 24.

6.30 (*Pattern recognition: consecutive four equal numbers*) Write the following method that tests whether the array has four consecutive numbers with the same value.

```
public static boolean isConsecutiveFour(int[] values)
```

Write a test program that prompts the user to enter a series of integers and displays true if the series contains four consecutive numbers with the same value. Otherwise, display false. Your program should first prompt the user to enter the input size—i.e., the number of values in the series.

This page intentionally left blank

CHAPTER 7

MULTIDIMENSIONAL ARRAYS

Objectives

- To give examples of representing data using two-dimensional arrays (§7.1).
- To declare variables for two-dimensional arrays, create arrays, and access array elements in a two-dimensional array using row and column indexes (§7.2).
- To program common operations for two-dimensional arrays (displaying arrays, summing all elements, finding min and max elements, and random shuffling) (§7.3).
- To pass two-dimensional arrays to methods (§7.4).
- To write a program for grading multiple-choice questions using two-dimensional arrays (§7.5).
- To solve the closest-pair problem using two-dimensional arrays (§7.6).
- To check a Sudoku solution using two-dimensional arrays (§7.7).
- To use multidimensional arrays (§7.8).



7.1 Introduction

problem

The preceding chapter introduced how to use one-dimensional arrays to store linear collections of elements. You can use a two-dimensional array to store a matrix or a table. For example, the following table that describes the distances between the cities can be stored using a two-dimensional array.

Distance Table (in miles)

	Chicago	Boston	New York	Atlanta	Miami	Dallas	Houston
Chicago	0	983	787	714	1375	967	1087
Boston	983	0	214	1102	1763	1723	1842
New York	787	214	0	888	1549	1548	1627
Atlanta	714	1102	888	0	661	781	810
Miami	1375	1763	1549	661	0	1426	1187
Dallas	967	1723	1548	781	1426	0	239
Houston	1087	1842	1627	810	1187	239	0

7.2 Two-Dimensional Array Basics

How do you declare a variable for two-dimensional arrays? How do you create a two-dimensional array? How do you access elements in a two-dimensional array? This section addresses these issues.

7.2.1 Declaring Variables of Two-Dimensional Arrays and Creating Two-Dimensional Arrays

Here is the syntax for declaring a two-dimensional array:

```
elementType[][] arrayRefVar;
```

or

```
elementType arrayRefVar[][]; // Allowed, but not preferred
```

As an example, here is how you would declare a two-dimensional array variable `matrix` of `int` values:

```
int[][] matrix;
```

or

```
int matrix[][]; // This style is allowed, but not preferred
```

You can create a two-dimensional array of `5`-by-`5` `int` values and assign it to `matrix` using this syntax:

```
matrix = new int[5][5];
```

Two subscripts are used in a two-dimensional array, one for the row and the other for the column. As in a one-dimensional array, the index for each subscript is of the `int` type and starts from `0`, as shown in Figure 7.1(a).

To assign the value `7` to a specific element at row `2` and column `1`, as shown in Figure 7.1(b), you can use the following:

```
matrix[2][1] = 7;
```

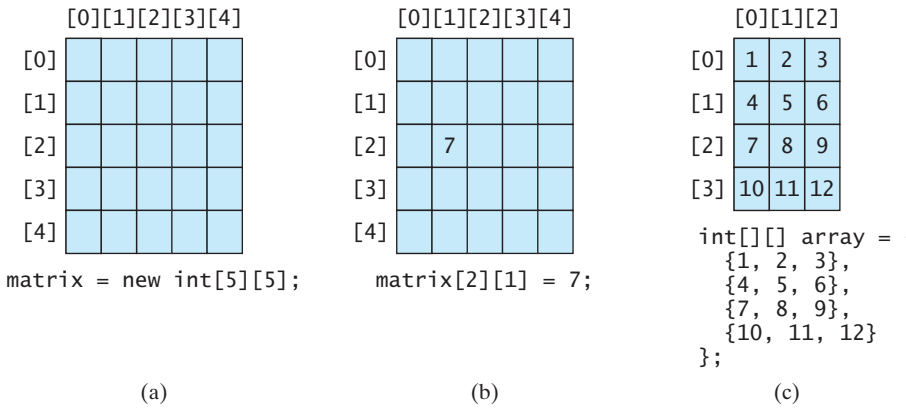


FIGURE 7.1 The index of each subscript of a two-dimensional array is an `int` value, starting from `0`.



Caution

It is a common mistake to use `matrix[2, 1]` to access the element at row `2` and column `1`. In Java, each subscript must be enclosed in a pair of square brackets.

You can also use an array initializer to declare, create, and initialize a two-dimensional array. For example, the following code in (a) creates an array with the specified initial values, as shown in Figure 7.1(c). This is equivalent to the code in (b).

```
int[][] array = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

(a)

Equivalent

```
int[][] array = new int[4][3];
array[0][0] = 1; array[0][1] = 2; array[0][2] = 3;
array[1][0] = 4; array[1][1] = 5; array[1][2] = 6;
array[2][0] = 7; array[2][1] = 8; array[2][2] = 9;
array[3][0] = 10; array[3][1] = 11; array[3][2] = 12;
```

(b)

7.2.2 Obtaining the Lengths of Two-Dimensional Arrays

A two-dimensional array is actually an array in which each element is a one-dimensional array. The length of an array `x` is the number of elements in the array, which can be obtained using `x.length`. `x[0]`, `x[1]`, ..., and `x[x.length-1]` are arrays. Their lengths can be obtained using `x[0].length`, `x[1].length`, ..., and `x[x.length-1].length`.

For example, suppose `x = new int[3][4]`, `x[0]`, `x[1]`, and `x[2]` are one-dimensional arrays and each contains four elements, as shown in Figure 7.2. `x.length` is `3`, and `x[0].length`, `x[1].length`, and `x[2].length` are `4`.

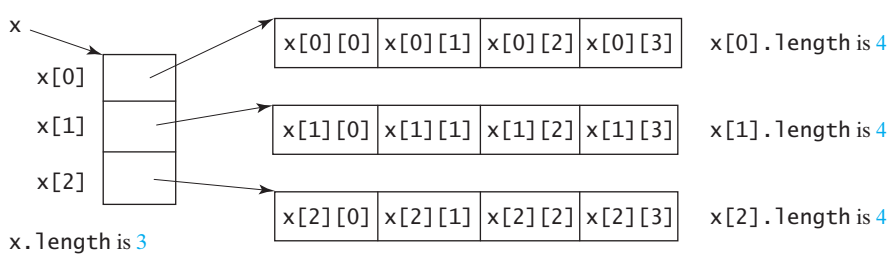
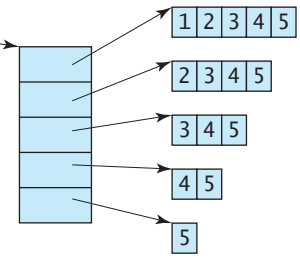


FIGURE 7.2 A two-dimensional array is a one-dimensional array in which each element is another one-dimensional array.

7.2.3 Ragged Arrays

Each row in a two-dimensional array is itself an array. Thus the rows can have different lengths. An array of this kind is known as a *ragged array*. Here is an example of creating a ragged array:

```
int[][] triangleArray = {
  {1, 2, 3, 4, 5},
  {2, 3, 4, 5},
  {3, 4, 5},
  {4, 5},
  {5}
};
```



As can be seen, `triangleArray[0].length` is 5, `triangleArray[1].length` is 4, `triangleArray[2].length` is 3, `triangleArray[3].length` is 2, and `triangleArray[4].length` is 1.

If you don't know the values in a ragged array in advance, but know the sizes, say the same as before, you can create a ragged array using the syntax that follows:

```
int[][] triangleArray = new int[5][];
triangleArray[0] = new int[5];
triangleArray[1] = new int[4];
triangleArray[2] = new int[3];
triangleArray[3] = new int[2];
triangleArray[4] = new int[1];
```

You can now assign values to the array. For example,

```
triangleArray[0][3] = 50;
triangleArray[4][0] = 45;
```



Note

The syntax `new int[5][]` for creating an array requires the first index to be specified. The syntax `new int[][]` would be wrong.

7.3 Processing Two-Dimensional Arrays

Suppose an array `matrix` is created as follows:

```
int[][] matrix = new int[10][10];
```

Here are some examples of processing two-dimensional arrays:

1. (*Initializing arrays with input values*) The following loop initializes the array with user input values:

```
java.util.Scanner input = new Scanner(System.in);
System.out.println("Enter " + matrix.length + " rows and " +
  matrix[0].length + " columns: ");
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
    matrix[row][column] = input.nextInt();
  }
}
```

2. (*Initializing arrays with random values*) The following loop initializes the array with random values between 0 and 99:

```
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
```

```
      matrix[row][column] = (int)(Math.random() * 100);
    }
  }
```

3. (*Printing arrays*) To print a two-dimensional array, you have to print each element in the array using a loop like the following:

```
for (int row = 0; row < matrix.length ; row++) {
  for (int column = 0; column < matrix[row].length ; column++) {
    System.out.print(matrix[row][column] + " ");
  }

  System.out.println();
}
```

4. (*Summing all elements*) Use a variable named `total` to store the sum. Initially `total` is `0`. Add each element in the array to `total` using a loop like this:

```
int total = 0;
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
    total += matrix[row][column];
  }
}
```

5. (*Summing elements by column*) For each column, use a variable named `total` to store its sum. Add each element in the column to `total` using a loop like this:

```
for (int column = 0; column < matrix[0].length; column++) {
  int total = 0;
  for (int row = 0; row < matrix.length; row++)
    total += matrix[row][column];
  System.out.println("Sum for column " + column + " is " +
    total);
}
```

6. (*Which row has the largest sum?*) Use variables `maxRow` and `indexOfMaxRow` to track the largest sum and index of the row. For each row, compute its sum and update `maxRow` and `indexOfMaxRow` if the new sum is greater.



Video Note
find the row with the largest sum

```
int maxRow = 0;
int indexOfMaxRow = 0;

// Get sum of the first row in maxRow
for (int column = 0; column < matrix[0].length; column++) {
  maxRow += matrix[0][column];
}

for (int row = 1; row < matrix.length; row++) {
  int totalOfThisRow = 0;
  for (int column = 0; column < matrix[row].length; column++)
    totalOfThisRow += matrix[row][column];

  if (totalOfThisRow > maxRow) {
    maxRow = totalOfThisRow;
    indexOfMaxRow = row;
  }
}

System.out.println("Row " + indexOfMaxRow
  + " has the maximum sum of " + maxRow);
```

7. (*Random shuffling*) Shuffling the elements in a one-dimensional array was introduced in §6.2.6. How do you shuffle all the elements in a two-dimensional array? To accomplish this, for each element `matrix[i][j]`, randomly generate indices `i1` and `j1` and swap `matrix[i][j]` with `matrix[i1][j1]`, as follows:

```
for (int i = 0; i < matrix.length; i++) {
  for (int j = 0; j < matrix[i].length; j++) {
    int i1 = (int)(Math.random() * matrix.length);
    int j1 = (int)(Math.random() * matrix[i].length);

    // Swap matrix[i][j] with matrix[i1][j1]
    int temp = matrix[i][j];
    matrix[i][j] = matrix[i1][j1];
    matrix[i1][j1] = temp;
  }
}
```

7.4 Passing Two-Dimensional Arrays to Methods

You can pass a two-dimensional array to a method just as you pass a one-dimensional array. Listing 7.1 gives an example with a method that returns the sum of all the elements in a matrix.

LISTING 7.1 PassTwoDimensionalArray.java

```
1  import java.util.Scanner;
2
3  public class PassTwoDimensionalArray {
4    public static void main(String[] args) {
5      // Create a Scanner
6      Scanner input = new Scanner(System.in);
7
8      // Enter array values
9      int[][] m = new int[3][4];
10     System.out.println("Enter " + m.length + " rows and "
11       + m[0].length + " columns: ");
12     for (int i = 0; i < m.length; i++)
13       for (int j = 0; j < m[i].length; j++)
14         m[i][j] = input.nextInt();
15
16     // Display result
17     System.out.println("\nSum of all elements is " + sum(m) );
18   }
19
20   public static int sum(int[][] m) {
21     int total = 0;
22     for (int row = 0; row < m.length; row++) {
23       for (int column = 0; column < m[row].length; column++) {
24         total += m[row][column];
25       }
26     }
27
28     return total;
29   }
30 }
```

pass array

```
Enter 3 rows and 4 columns:
1 2 3 4 Enter
5 6 7 8 Enter
9 10 11 12 Enter
Sum of all elements is 78
```



The method `sum` (line 20) has a two-dimensional array argument. You can obtain the number of rows using `m.length` (line 22) and the number of columns in a specified row using `m[row].column` (line 23).

7.5 Problem: Grading a Multiple-Choice Test



Video Note
Grade multiple-choice test

The problem is to write a program that grades multiple-choice tests. Suppose there are eight students and ten questions, and the answers are stored in a two-dimensional array. Each row records a student's answers to the questions, as shown in the following array.

Students' Answers to the Questions:

	0	1	2	3	4	5	6	7	8	9
Student 0	A	B	A	C	C	D	E	E	A	D
Student 1	D	B	A	B	C	A	E	E	A	D
Student 2	E	D	D	A	C	B	E	E	A	D
Student 3	C	B	A	E	D	C	E	E	A	D
Student 4	A	B	D	C	C	D	E	E	A	D
Student 5	B	B	E	C	C	D	E	E	A	D
Student 6	B	B	A	C	C	D	E	E	A	D
Student 7	E	B	E	C	C	D	E	E	A	D

The key is stored in a one-dimensional array:

Key to the Questions:

	0	1	2	3	4	5	6	7	8	9
Key	D	B	D	C	C	D	A	E	A	D

Your program grades the test and displays the result. It compares each student's answers with the key, counts the number of correct answers, and displays it. Listing 7.2 gives the program.

LISTING 7.2 GradeExam.java

```
1  public class GradeExam {
2    /** Main method */
3    public static void main(String[] args) {
4      // Students' answers to the questions
5      char[][] answers = {                                          2-D array
6        {'A', 'B', 'A', 'C', 'C', 'D', 'E', 'E', 'A', 'D'},
7        {'D', 'B', 'A', 'B', 'C', 'A', 'E', 'E', 'A', 'D'},
8        {'E', 'D', 'D', 'A', 'C', 'B', 'E', 'E', 'A', 'D'},
9        {'C', 'B', 'A', 'E', 'D', 'C', 'E', 'E', 'A', 'D'},
10       {'A', 'B', 'D', 'C', 'C', 'D', 'E', 'E', 'A', 'D'},
11       {'B', 'B', 'E', 'C', 'C', 'D', 'E', 'E', 'A', 'D'},
12       {'B', 'B', 'A', 'C', 'C', 'D', 'E', 'E', 'A', 'D'},
13       {'E', 'B', 'E', 'C', 'C', 'D', 'E', 'E', 'A', 'D'}};
```

1-D array

compare with key

```
14
15      // Key to the questions
16      char[] keys = {'D', 'B', 'D', 'C', 'C', 'D', 'A', 'E', 'A', 'D'};
17
18      // Grade all answers
19      for (int i = 0; i < answers.length ; i++) {
20        // Grade one student
21        int correctCount = 0;
22        for (int j = 0; j < answers[i].length ; j++) {
23          if (answers[i][j] == keys[j] )
24            correctCount++;
25        }
26
27        System.out.println("Student " + i + "'s correct count is " +
28          correctCount);
29      }
30    }
31 }
```

```
Student 0's correct count is 7
Student 1's correct count is 6
Student 2's correct count is 5
Student 3's correct count is 4
Student 4's correct count is 8
Student 5's correct count is 7
Student 6's correct count is 7
Student 7's correct count is 7
```

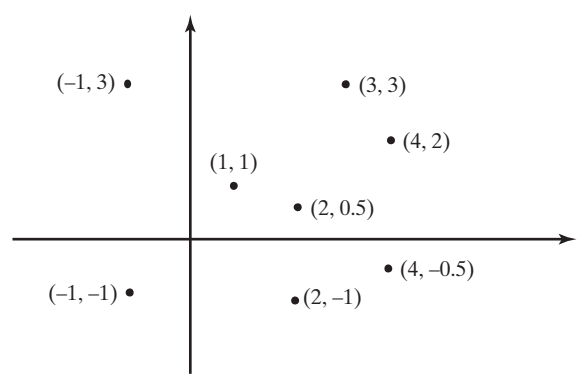
The statement in lines 5–13 declares, creates, and initializes a two-dimensional array of characters and assigns the reference to **answers** of the **char[][]** type.

The statement in line 16 declares, creates, and initializes an array of **char** values and assigns the reference to **keys** of the **char[]** type.

Each row in the array **answers** stores a student's answer, which is graded by comparing it with the key in the array **keys**. The result is displayed immediately after a student's answer is graded.

7.6 Problem: Finding a Closest Pair

The GPS navigation system is becoming increasingly popular. The system uses the graph and geometric algorithms to calculate distances and map a route. This section presents a geometric problem for finding a closest pair of points.



	x	y
0	–1	3
1	–1	–1
2	1	1
3	2	0.5
4	2	–1
5	3	3
6	4	2
7	4	–0.5

FIGURE 7.3 Points can be represented in a two-dimensional array.

Given a set of points, the closest-pair problem is to find the two points that are nearest to each other. In Figure 7.3, for example, points `(1, 1)` and `(2, 0.5)` are closest to each other. There are several ways to solve this problem. An intuitive approach is to compute the distances between all pairs of points and find the one with the minimum distance, as implemented in Listing 7.3.

LISTING 7.3 FindNearestPoints.java

```
1 import java.util.Scanner;
2
3 public class FindNearestPoints {
4   public static void main(String[] args) {
5     Scanner input = new Scanner(System.in);
6     System.out.print("Enter the number of points: ");
7     int numberOfPoints = input.nextInt();                          number of points
8
9     // Create an array to store points
10    double[][] points = new double[numberOfPoints][2];             2-D array
11    System.out.print("Enter " + numberOfPoints + " points: ");
12    for (int i = 0; i < points.length; i++) {                      read points
13      points[i][0] = input.nextDouble();
14      points[i][1] = input.nextDouble();
15    }
16
17    // p1 and p2 are the indices in the points array
18    int p1 = 0, p2 = 1; // Initial two points                      track two points
19    double shortestDistance = distance(points[p1][0], points[p1][1],   track shortestDistance
20      points[p2][0], points[p2][1]); // Initialize shortestDistance
21
22    // Compute distance for every two points
23    for (int i = 0; i < points.length; i++) {                      for each point i
24      for (int j = i + 1; j < points.length; j++) {                for each point j
25        double distance = distance(points[i][0], points[i][1],     distance between i and j
26          points[j][0], points[j][1]); // Find distance            distance between two points
27
28        if (shortestDistance > distance) {
29          p1 = i; // Update p1
30          p2 = j; // Update p2
31          shortestDistance = distance; // Update shortestDistance  update shortestDistance
32        }
33      }
34    }
35
36    // Display result
37    System.out.println("The closest two points are " +
38      "(" + points[p1][0] + ", " + points[p1][1] + ") and (" +
39      points[p2][0] + ", " + points[p2][1] + ")");
40  }
41
42  /** Compute the distance between two points (x1, y1) and (x2, y2)*/
43  public static double distance(
44      double x1, double y1, double x2, double y2) {
45    return Math.sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
46  }
47 }
```

```
Enter the number of points: 8 Enter
Enter 8 points: -1 3  -1 -1  1 1  2 0.5  2 -1  3 3  4 2 4 -0.5 Enter
The closest two points are (1, 1) and (2, 0.5)
```



The program prompts the user to enter the number of points (lines 6–7). The points are read from the console and stored in a two-dimensional array named `points` (lines 12–15). The program uses variable `shortestDistance` (line 19) to store the distance between two nearest points, and the indices of these two points in the `points` array are stored in `p1` and `p2` (line 18).

For each point at index `i`, the program computes the distance between `points[i]` and `points[j]` for all `j > i` (lines 23–34). Whenever a shorter distance is found, the variable `shortestDistance` and `p1` and `p2` are updated (lines 28–32).

The distance between two points `(x1, y1)` and `(x2, y2)` can be computed using the formula $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ (lines 43–46).

The program assumes that the plane has at least two points. You can easily modify the program to handle the case if the plane has zero or one point.

multiple closest pairs

Note that there might be more than one closest pair of points with the same minimum distance. The program finds one such pair. You may modify the program to find all closest pairs in Programming Exercise 7.8.



Tip

input file

It is cumbersome to enter all points from the keyboard. You may store the input in a file, say FindNearestPoints.txt, and compile and run the program using the following command:

```
java FindNearestPoints < FindNearestPoints.txt
```

7.7 Problem: Sudoku



Video Note
Sudoku

This book teaches how to program using a wide variety of problems with various levels of difficulty. We use simple, short, and stimulating examples to introduce programming and problem-solving techniques and use interesting and challenging examples to motivate students. This section presents an interesting problem of a sort that appears in the newspaper every day. It is a number-placement puzzle, commonly known as *Sudoku*. This is a very challenging problem. To make it accessible to the novice, this section presents a solution to a simplified version of the Sudoku problem, which is to verify whether a solution is correct. The complete solution for solving the Sudoku problem is presented in Supplement VII.A.

fixed cells
free cells

Sudoku is a 9 × 9 grid divided into smaller 3 × 3 boxes (also called regions or blocks), as shown in Figure 7.4(a). Some cells, called *fixed cells*, are populated with numbers from `1` to `9`. The objective is to fill the empty cells, also called *free cells*, with numbers `1` to `9` so that every row, every column, and every 3 × 3 box contains the numbers `1` to `9`, as shown in Figure 7.4(b).

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6							
			4	1	9			5
				8			7	9

(a) Puzzle

Solution →

5	3	*4*	*6*	7	*8*	9	*1*	2
6	*7*	*2*	1	9	5	*3*	*4*	*8*
1	9	8	*3*	*4*	*2*	*5*	6	*7*
8	*5*	*9*	*7*	6	*1*	*4*	*2*	3
4	*2*	*6*	8	*5*	3	*7*	*9*	1
7	*1*	*3*	*9*	2	*4*	*8*	*5*	6
9	6	*1*	*5*	*3*	*7*	*2*	*8*	*4*
2	*8*	*7*	4	1	9	*6*	*3*	5
3	*4*	*5*	*2*	8	*6*	*1*	7	9

(b) Solution

FIGURE 7.4 The Sudoku puzzle in (a) is solved in (b).

For convenience, we use value 0 to indicate a free cell, as shown in Figure 7.5(a). The grid can be naturally represented using a two-dimensional array, as shown in Figure 7.5(a). representing a grid

5	3	0	0	7	0	0	0	0
6	0	0	1	9	5	0	0	0
0	9	8	0	0	0	0	6	0
8	0	0	0	6	0	0	0	3
4	0	0	8	0	3	0	0	1
7	0	0	0	2	0	0	0	6
0	6	0	0	0	0	0	0	0
0	0	0	4	1	9	0	0	5
0	0	0	0	8	0	0	7	9

(a)

```
int[][] grid =
  {{5, 3, 0, 0, 7, 0, 0, 0, 0},
   {6, 0, 0, 1, 9, 5, 0, 0, 0},
   {0, 9, 8, 0, 0, 0, 0, 6, 0},
   {8, 0, 0, 0, 6, 0, 0, 0, 3},
   {4, 0, 0, 8, 0, 3, 0, 0, 1},
   {7, 0, 0, 0, 2, 0, 0, 0, 6},
   {0, 6, 0, 0, 0, 0, 2, 8, 0},
   {0, 0, 0, 4, 1, 9, 0, 0, 5},
   {0, 0, 0, 0, 8, 0, 0, 7, 9}
  };
```

(b)

FIGURE 7.5 A grid can be represented using a two-dimensional array.

To find a solution for the puzzle we must replace each 0 in the grid with an appropriate number from 1 to 9. For the solution in Figure 7.4(b), the grid should be as shown in Figure 7.6.

```
A solution grid is
  {{5, 3, 4, 6, 7, 8, 9, 1, 2},
   {6, 7, 2, 1, 9, 5, 3, 4, 8},
   {1, 9, 8, 3, 4, 2, 5, 6, 7},
   {8, 5, 9, 7, 6, 1, 4, 2, 3},
   {4, 2, 6, 8, 5, 3, 7, 9, 1},
   {7, 1, 3, 9, 2, 4, 8, 5, 6},
   {9, 6, 1, 5, 3, 7, 2, 8, 4},
   {2, 8, 7, 4, 1, 9, 6, 3, 5},
   {3, 4, 5, 2, 8, 6, 1, 7, 9}
  };
```

FIGURE 7.6 A solution is stored in `grid`.

A simplified version of the Sudoku problem is to check the validity of a solution. The program in Listing 7.4 prompts the user to enter a solution and reports whether it is valid.

LISTING 7.4 CheckSudokuSolution.java

```
1  import java.util.Scanner;
2
3  public class CheckSudokuSolution {
4    public static void main(String[] args) {
5      // Read a Sudoku solution
6      int[][] grid = readASolution();                          read input
7
8      System.out.println(isValid(grid) ? "Valid solution" :    solution valid?
9        "Invalid solution");
10   }
11
12   /** Read a Sudoku solution from the console */
13   public static int[][] readASolution() {                    read solution
14     // Create a Scanner
15     Scanner input = new Scanner(System.in);
```

```
16
17    System.out.println("Enter a Sudoku puzzle solution:");
18    int[][] grid = new int[9][9];
19    for (int i = 0; i < 9; i++)
20      for (int j = 0; j < 9; j++)
21        grid[i][j] = input.nextInt();
22
23    return grid;
24  }
25
26  /** Check whether a solution is valid */
27  public static boolean isValid(int[][] grid) {
28    // Check whether each row has numbers 1 to 9
29    for (int i = 0; i < 9; i++)
30      if (!is1To9(grid[i]) ) // If grid[i] does not contain 1 to 9
31        return false;
32
33    // Check whether each column has numbers 1 to 9
34    for (int j = 0; j < 9; j++) {
35      // Obtain a column in the one-dimensional array
36      int[] column = new int[9];
37      for (int i = 0; i < 9; i++) {
38        column[i] = grid[i][j];
39      }
40
41      if (!is1To9(column) ) // If column does not contain 1 to 9
42        return false;
43    }
44
45    // Check whether each 3-by-3 box has numbers 1 to 9
46    for (int i = 0; i < 3; i++) {
47      for (int j = 0; j < 3; j++) {
48        // The starting element in a small 3-by-3 box
49        int k = 0;
50        int[] list = new int[9]; // Get all numbers in the box to list
51        for (int row = i * 3; row < i * 3 + 3; row ++)
52          for (int column = j * 3; column < j * 3 + 3; column++)
53            list[k++] = grid[row][column];
54
55        if (!is1To9(list) ) // If list does not contain 1 to 9
56          return false;
57      }
58    }
59
60    return true; // The fixed cells are valid
61  }
62
63  /** Check whether the one-dimensional array contains 1 to 9 */
64  public static boolean is1To9(int[] list) {
65    // Make a copy of the array
66    int[] temp = new int[list.length];
67    System.arraycopy(list, 0, temp, 0, list.length);
68
69    // Sort the array
70    java.util.Arrays.sort(temp);
71
72    // Check whether the list contains 1, 2, 3, ..., 9
73    for (int i = 0; i < 9; i++)
```

check solution

check rows

check columns

check small boxes

all valid

contains 1 to 9 ?

copy of array

sort array

check 1 to 9

```
74        if (temp[i] != i + 1)
75          return false;
76
77      return true; // The list contains exactly 1 to 9
78    }
79  }
```

```
Enter a Sudoku puzzle solution:
9 6 3 1 7 4 2 5 8 Enter
1 7 8 3 2 5 6 4 9 Enter
2 5 4 6 8 9 7 3 1 Enter
8 2 1 4 3 7 5 9 6 Enter
4 9 6 8 5 2 3 1 7 Enter
7 3 5 9 6 1 8 2 4 Enter
5 8 9 7 1 3 4 6 2 Enter
3 1 7 2 4 6 9 8 5 Enter
6 4 2 5 9 8 1 7 3 Enter
Valid solution
```



The program invokes the `readASolution()` method (line 6) to read a Sudoku solution and return a two-dimensional array representing a Sudoku grid.

`isValid` method

The `isValid(grid)` method (lines 27–61) checks whether every row contains numbers `1` to `9` (lines 29–31). `grid` is a two-dimensional array. `grid[i]` is a one-dimensional array for the *i*th row. Invoking `isTo9(grid[i])` returns `true` if the row `grid[i]` contains exactly numbers from `1` to `9` (line 30).

check rows

To check whether each column in `grid` has numbers `1` to `9`, get a column into a one-dimensional array (lines 36–39) and invoke the `isTo9` method to check whether it has `1` to `9` (line 41).

check columns

To check whether each small 3 × 3 box in `grid` has numbers `1` to `9`, get a box into a one-dimensional array (lines 49–53) and invoke the `isTo9` method to check whether it has `1` to `9` (line 55).

check small boxes

How do you locate all the cells in the same box? First, locate the starting cells of the 3 × 3 boxes. They are at `(3i, 3j)` for `i = 0, 1, 2` and `j = 0, 1, 2`, as illustrated in Figure 7.7.

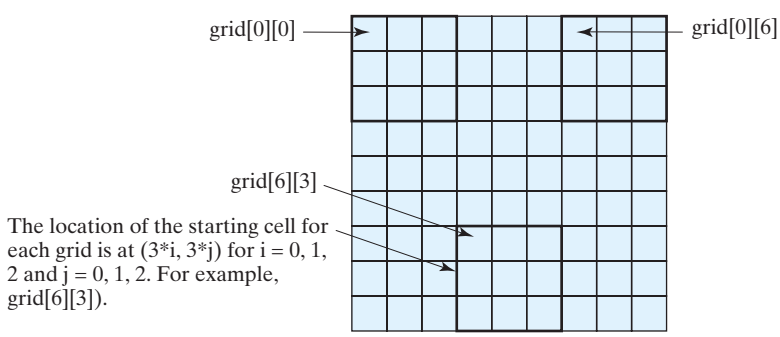


FIGURE 7.7 The location of the first cell in a 3 × 3 box determines the locations of other cells in the box.

With this observation, you can easily identify all the cells in the box. Suppose `grid[r][c]` is the starting cell of a 3 × 3 box. The cells in the box can be traversed in a nested loop as follows:

```
// Get all cells in a 3-by-3 box starting at grid[r][c]
for (int row = r; row < r + 3; row++)
  for (int column = c; column < c + 3; column++)
    // grid[row][column] is in the box
```

All the numbers in a small box are collected into a one-dimensional array `list` (line 53), and invoking `isITo9(list)` checks whether `list` contains numbers `1` to `9` (line 55).

`isITo9` method

The `isITo9(list)` method (lines 64–78) checks whether array `list` contains exactly numbers `1` to `9`. It first copies `list` to a new array `temp`, then sorts `temp`. Note that if you sort `list`, the contents of `grid` will be changed. After `temp` is sorted, the numbers in `temp` should be `1`, `2`, ..., `9`, if `temp` contains exactly `1` to `9`. The loop in lines 73–75 checks whether this is the case.

input file

It is cumbersome to enter 81 numbers from the console. When you test the program, you may store the input in a file, say CheckSudokuSolution.txt, and run the program using the following command:

```
java CheckSudokuSoluton < CheckSudokuSoluton.txt
```

7.8 Multidimensional Arrays

In the preceding section, you used a two-dimensional array to represent a matrix or a table. Occasionally, you will need to represent *n*-dimensional data structures. In Java, you can create *n*-dimensional arrays for any integer *n*.

The way to declare two-dimensional array variables and create two-dimensional arrays can be generalized to declare *n*-dimensional array variables and create *n*-dimensional arrays for $n >= 3$. For example, the following syntax declares a three-dimensional array variable `scores`, creates an array, and assigns its reference to `scores`.

```
double[][][] data = new double[10][24][2];
```

A multidimensional array is actually an array in which each element is another array. A three-dimensional array consists of an array of two-dimensional arrays, each of which is an array of one-dimensional arrays. For example, suppose `x = new int[2][2][5]`, `x[0]` and `x[1]` are two-dimensional arrays. `X[0][0]`, `x[0][1]`, `x[1][0]`, and `x[1][1]` are one-dimensional arrays and each contains five elements. `x.length` is `2`, `x[0].length` and `x[1].length` are `2`, and `X[0][0].length`, `x[0][1].length`, `x[1][0].length`, and `x[1][1].length` are `5`.

7.8.1 Problem: Daily Temperature and Humidity

Suppose a meteorology station records the temperature and humidity at each hour of every day and stores the data for the past ten days in a text file named weather.txt. Each line of the file consists of four numbers that indicate the day, hour, temperature, and humidity. The contents of the file may look like the one in (a):

```
1 1 76.4 0.92
1 2 77.7 0.93
...
10 23 97.7 0.71
10 24 98.7 0.74
```

(a)

```
10 24 98.7 0.74
1 2 77.7 0.93
...
10 23 97.7 0.71
1 1 76.4 0.92
```

(b)

Note that the lines in the file are not necessary in order. For example, the file may appear as shown in (b).

Your task is to write a program that calculates the average daily temperature and humidity for the 10 days. You can use the input redirection to read the file and store the data in a three-dimensional array, named data. The first index of data ranges from 0 to 9 and represents 10 days, the second index ranges from 0 to 23 and represents 24 hours, and the third index ranges from 0 to 1 and represents temperature and humidity, respectively. Note that the days are numbered from 1 to 10 and hours from 1 to 24 in the file. Since the array index starts from 0, data[0][0][0] stores the temperature in day 1 at hour 1 and data[9][23][1] stores the humidity in day 10 at hour 24.

The program is given in Listing 7.5.

LISTING 7.5 Weather.java

```
1 import java.util.Scanner;
2
3 public class Weather {
4   public static void main(String[] args) {
5     final int NUMBER_OF_DAYS = 10;
6     final int NUMBER_OF_HOURS = 24;
7     double[][][] data
8       = new double[NUMBER_OF_DAYS][NUMBER_OF_HOURS][2];
9
10    Scanner input = new Scanner(System.in);
11    // Read input using input redirection from a file
12    for (int k = 0; k < NUMBER_OF_DAYS * NUMBER_OF_HOURS; k++) {
13      int day = input.nextInt();
14      int hour = input.nextInt();
15      double temperature = input.nextDouble();
16      double humidity = input.nextDouble();
17      data[day - 1][hour - 1][0] = temperature;
18      data[day - 1][hour - 1][1] = humidity;
19    }
20
21    // Find the average daily temperature and humidity
22    for (int i = 0; i < NUMBER_OF_DAYS; i++) {
23      double dailyTemperatureTotal = 0, dailyHumidityTotal = 0;
24      for (int j = 0; j < NUMBER_OF_HOURS; j++) {
25        dailyTemperatureTotal += data[i][j][0];
26        dailyHumidityTotal += data[i][j][1];
27      }
28
29      // Display result
30      System.out.println("Day  " + i + "'s average temperature is "
31        + dailyTemperatureTotal / NUMBER_OF_HOURS);
32      System.out.println("Day  " + i + "'s average humidity is "
33        + dailyHumidityTotal / NUMBER_OF_HOURS);
34    }
35  }
35 }
```

three-dimensional array

```
Day  0's average temperature is 77.7708
Day  0's average humidity is 0.929583
Day  1's average temperature is 77.3125
Day  1's average humidity is 0.929583
...
Day  9's average temperature is 79.3542
Day  9's average humidity is 0.9125
```



You can use the following command to run the program:

```
java Weather < Weather.txt
```

A three-dimensional array for storing temperature and humidity is created in line 8. The loop in lines 12–19 reads the input to the array. You can enter the input from the keyboard, but doing so will be awkward. For convenience, we store the data in a file and use the input redirection to read the data from the file. The loop in lines 24–27 adds all temperatures for each hour in a day to `dailyTemperatureTotal` and all humidity for each hour to `dailyHumidityTotal`. The average daily temperature and humidity are displayed in lines 30–33.

7.8.2 Problem: Guessing Birthdays

Listing 3.3, GuessBirthday.java, gives a program that guesses a birthday. The program can be simplified by storing the numbers in five sets in a three-dimensional array, and it prompts the user for the answers using a loop, as shown in Listing 7.6. The sample run of the program can be the same as shown in Listing 3.3.

LISTING 7.6 GuessBirthdayUsingArray.java

three-dimensional array

Set i

```
1  import java.util.Scanner;
2
3  public class GuessBirthdayUsingArray {
4    public static void main(String[] args) {
5      int day = 0; // Day to be determined
6      int answer;
7
8      int[][][] dates = {
9        {{ 1,  3,  5,  7},
10        { 9, 11, 13, 15},
11        {17, 19, 21, 23},
12        {25, 27, 29, 31}},
13       {{ 2,  3,  6,  7},
14        {10, 11, 14, 15},
15        {18, 19, 22, 23},
16        {26, 27, 30, 31}},
17       {{ 4,  5,  6,  7},
18        {12, 13, 14, 15},
19        {20, 21, 22, 23},
20        {28, 29, 30, 31}},
21       {{ 8,  9, 10, 11},
22        {12, 13, 14, 15},
23        {24, 25, 26, 27},
24        {28, 29, 30, 31}},
25       {{16, 17, 18, 19},
26        {20, 21, 22, 23},
27        {24, 25, 26, 27},
28        {28, 29, 30, 31}}};
29
30     // Create a Scanner
31     Scanner input = new Scanner(System.in);
32
33     for (int i = 0; i < 5; i++) {
34       System.out.println("Is your birthday in Set" + (i + 1) + "?");
35       for (int j = 0; j < 4; j++) {
36         for (int k = 0; k < 4; k++)
37           System.out.printf("%4d", dates[i][j][k]);
38         System.out.println();
39       }
```

```
40
41        System.out.print("\nEnter 0 for No and 1 for Yes: ");
42        answer = input.nextInt();
43
44        if (answer == 1)
45          day += dates[i][0][0] ;                    add to Set i
46      }
47
48      System.out.println("Your birth day is " + day);
49    }
50 }
```

A three-dimensional array `dates` is created in Lines 8–28. This array stores five sets of numbers. Each set is a `4`-by-`4` two-dimensional array.

The loop starting from line 33 displays the numbers in each set and prompts the user to answer whether the birthday is in the set (lines 41–42). If the day is in the set, the first number (`dates[i][0][0]`) in the set is added to variable `day` (line 45).

Chapter Summary

1. A two-dimensional array can be used to store a table.
2. A variable for two-dimensional arrays can be declared using the syntax: `elementType[][] arrayVar`.
3. A two-dimensional array can be created using the syntax: `new elementType[ROW_SIZE][COLUMN_SIZE]`.
4. Each element in a two-dimensional array is represented using the syntax: `arrayVar[rowIndex][columnIndex]`.
5. You can create and initialize a two-dimensional array using an array initializer with the syntax: `elementType[][] arrayVar = {{row values}, ..., {row values}}`.
6. You can use arrays of arrays to form multidimensional arrays. For example, a variable for three-dimensional arrays can be declared as `elementType[][][] arrayVar` and a three-dimensional array can be created using `new elementType[size1][size2][size3]`.

Review Questions

7.1 Declare and create a `4`-by-`5` `int` matrix.

7.2 Can the rows in a two-dimensional array have different lengths?

7.3 What is the output of the following code?

```
int[][] array = new int[5][6];
int[] x = {1, 2};
array[0] = x;
System.out.println("array[0][1] is " + array[0][1]);
```

7.4 Which of the following statements are valid array declarations?

```
int[][] r = new int[2];
int[] x = new int[];
int[][] y = new int[3][];
```

7.5 Why does the `is1To9` method need to copy `list` to `temp`? What happens if you replace the code in lines 66–70 in Listing 7.4 with the following code:

```
java.util.Arrays.sort(list);
```

7.6 Declare and create a $4 \times 6 \times 5$ `int` array.

Programming Exercises

7.1* (*Summing all the numbers in a matrix*) Write a method that sums all the integers in a matrix of integers using the following header:

```
public static double sumMatrix(int[][] m)
```

Write a test program that reads a 4-by-4 matrix and displays the sum of all its elements. Here is a sample run:



```
Enter a 4-by-4 matrix row by row:
1 2 3 4 Enter
5 6 7 8 Enter
9 10 11 12 Enter
13 14 15 16 Enter
Sum of the matrix is 136
```

7.2* (*Summing the major diagonal in a matrix*) Write a method that sums all the integers in the major diagonal in an $n \times n$ matrix of integers using the following header:

```
public static int sumMajorDiagonal(int[][] m)
```

Write a test program that reads a 4-by-4 matrix and displays the sum of all its elements on the major diagonal. Here is a sample run:



```
Enter a 4-by-4 matrix row by row:
1 2 3 4 Enter
5 6 7 8 Enter
9 10 11 12 Enter
13 14 15 16 Enter
Sum of the elements in the major diagonal is 34
```

7.3* (*Sorting students on grades*) Rewrite Listing 7.2, GradeExam.java, to display the students in increasing order of the number of correct answers.

7.4** (*Computing the weekly hours for each employee*) Suppose the weekly hours for all employees are stored in a two-dimensional array. Each row records an employee's seven-day work hours with seven columns. For example, the array shown below stores the work hours for eight employees. Write a program that displays employees and their total hours in decreasing order of the total hours.

	Su	M	T	W	H	F	Sa
Employee 0	2	4	3	4	5	8	8
Employee 1	7	3	4	3	3	4	4
Employee 2	3	3	4	3	3	2	2
Employee 3	9	3	4	7	3	4	1
Employee 4	3	5	4	3	6	3	8
Employee 5	3	4	4	6	3	4	4
Employee 6	3	7	4	8	3	8	4
Employee 7	6	3	5	9	2	7	9

7.5 (*Algebra: adding two matrices*) Write a method to add two matrices. The header of the method is as follows:

```
public static double[][] addMatrix(double[][] a, double[][] b)
```

In order to be added, the two matrices must have the same dimensions and the same or compatible types of elements. Let **c** be the resulting matrix. Each element c_{ij} is $a_{ij} + b_{ij}$. For example, for two 3×3 matrices **a** and **b**, **c** is

$$\begin{pmatrix} a_{11}\ a_{12}\ a_{13} \\ a_{21}\ a_{22}\ a_{23} \\ a_{31}\ a_{32}\ a_{33} \end{pmatrix} + \begin{pmatrix} b_{11}\ b_{12}\ b_{13} \\ b_{21}\ b_{22}\ b_{23} \\ b_{31}\ b_{32}\ b_{33} \end{pmatrix} = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & a_{13} + b_{13} \\ a_{21} + b_{21} & a_{22} + b_{22} & a_{23} + b_{23} \\ a_{31} + b_{31} & a_{32} + b_{32} & a_{33} + b_{33} \end{pmatrix}$$

Write a test program that prompts the user to enter two 3×3 matrices and displays their sum. Here is a sample run:



```
Enter matrix1: 1 2 3 4 5 6 7 8 9 ↵Enter
Enter matrix2: 0 2 4 1 4.5 2.2 1.1 4.3 5.2 ↵Enter
The matrices are added as follows
 1.0 2.0 3.0      0.0 2.0 4.0      1.0 4.0 7.0
 4.0 5.0 6.0  +   1.0 4.5 2.2  =   5.0 9.5 8.2
 7.0 8.0 9.0      1.1 4.3 5.2      8.1 12.3 14.2
```

7.6** (*Algebra: multiplying two matrices*) Write a method to multiply two matrices. The header of the method is as follows:



Video Note
Multiply two matrices

```
public static double[][] multiplyMatrix(double[][] a, double[][] b)
```

To multiply matrix **a** by matrix **b**, the number of columns in **a** must be the same as the number of rows in **b**, and the two matrices must have elements of the same or compatible types. Let **c** be the result of the multiplication. Assume the column size of matrix a is **n**. Each element c_{ij} is $a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + \ldots + a_{in} \times b_{nj}$. For example, for two 3×3 matrices **a** and **b**, **c** is

$$\begin{pmatrix} a_{11}\ a_{12}\ a_{13} \\ a_{21}\ a_{22}\ a_{23} \\ a_{31}\ a_{32}\ a_{33} \end{pmatrix} \times \begin{pmatrix} b_{11}\ b_{12}\ b_{13} \\ b_{21}\ b_{22}\ b_{23} \\ b_{31}\ b_{32}\ b_{33} \end{pmatrix} = \begin{pmatrix} c_{11}\ c_{12}\ c_{13} \\ c_{21}\ c_{22}\ c_{23} \\ c_{31}\ c_{32}\ c_{33} \end{pmatrix}$$

where $c_{ij} = a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + a_{i3} \times b_{3j}$.

Write a test program that prompts the user to enter two 3×3 matrices and displays their product. Here is a sample run:



```
Enter matrix1: 1 2 3 4 5 6 7 8 9 Enter
Enter matrix2: 0 2 4 1 4.5 2.2 1.1 4.3 5.2 Enter
The matrices are multiplied as follows:
 1 2 3      0 2.0 4.0      5.3 23.9 24
 4 5 6   *  1 4.5 2.2  =   11.6 56.3 58.2
 7 8 9      1.1 4.3 5.2    17.9 88.7 92.4
```

7.7* (*Points nearest to each other*) Listing 7.3 gives a program that finds two points in a two-dimensional space nearest to each other. Revise the program so that it finds two points in a three-dimensional space nearest to each other. Use a two-dimensional array to represent the points. Test the program using the following points:

```
double[][] points = {{-1, 0, 3}, {-1, -1, -1}, {4, 1, 1},
  {2, 0.5, 9}, {3.5, 2, -1}, {3, 1.5, 3}, {-1.5, 4, 2},
  {5.5, 4, -0.5}};
```

The formula for computing the distance between two points `(x1, y1, z1)` and `(x2, y2, z2)` is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$.

7.8** (*All closest pairs of points*) Revise Listing 7.3, FindNearestPoints.java, to find all closest pairs of points with same minimum distance.

7.9*** (*Game: playing a TicTacToe game*) In a game of TicTacToe, two players take turns marking an available cell in a `3` × `3` grid with their respective tokens (either X or O). When one player has placed three tokens in a horizontal, vertical, or diagonal row on the grid, the game is over and that player has won. A draw (no winner) occurs when all the cells on the grid have been filled with tokens and neither player has achieved a win. Create a program for playing TicTacToe.

The program prompts two players to enter X token and O token alternately. Whenever a token is entered, the program redisplays the board on the console and determines the status of the game (win, draw, or continue). Here is a sample run:



```
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
Enter a row (1, 2, or 3) for player X: 1 Enter
Enter a column (1, 2, or 3) for player X: 1 Enter
-------------
|   |   |   |
-------------
|   | X |   |
-------------
|   |   |   |
-------------
Enter a row (1, 2, or 3) for player O: 1 Enter
Enter a column (1, 2, or 3) for player O: 2 Enter
-------------
|   |   |   |
-------------
|   | X | O |
-------------
|   |   |   |
-------------
```

```
Enter a row (1, 2, or 3) for player X:
...
-------------
| X |   |   |
-------------
| O | X | O |
-------------
|   |   | X |
-------------
X player won
```

7.10* (*Game: TicTacToe board*) Write a program that randomly fills in 0s and 1s into a TicTacToe board, prints the board, and finds the rows, columns, or diagonals with all 0s or 1s. Use a two-dimensional array to represent a TicTacToe board. Here is a sample run of the program:

```
001
001
111
All 1s on row 2
All 1s on column 2
```

7.11** (*Game: nine heads and tails*) Nine coins are placed in a 3-by-3 matrix with some face up and some face down. You can represent the state of the coins using a 3-by-3 matrix with values **0** (head) and **1** (tail). Here are some examples:

```
0 0 0    1 0 1    1 1 0    1 0 1    1 0 0
0 1 0    0 0 1    1 0 0    1 1 0    1 1 1
0 0 0    1 0 0    0 0 1    1 0 0    1 1 0
```

Each state can also be represented using a binary number. For example, the preceding matrices correspond to the numbers

```
000010000 101001100 110100001 101110100 100111110
```

There are a total of **512** possibilities. So, you can use decimal numbers **0**, **1**, **2**, **3**, . . . , and **511** to represent all states of the matrix. Write a program that prompts the user to enter a number between **0** and **511** and displays the corresponding matrix with characters **H** and **T**. Here is a sample run:

```
Enter a number between 0 and 511: 7 Enter
H H H
H H H
T T T
```



The user entered **7**, which corresponds to **000000111**. Since **0** stands for **H** and **1** for **T**, the output is correct.

7.12** (*Financial application: computing tax*) Rewrite Listing 3.6, ComputeTax.java, using arrays. For each filing status, there are six tax rates. Each rate is applied to a certain amount of taxable income. For example, from the taxable income of $400,000 for a single filer, $8,350 is taxed at 10%, (33,950 – 8,350) at 15%, (82,250 – 33,950) at 25%, (171,550 – 82,550) at 28%, (372,550 – 82,250) at 33%,

and (400,000 – 372,950) at 36%. The six rates are the same for all filing statuses, which can be represented in the following array:

```
double[] rates = {0.10, 0.15, 0.25, 0.28, 0.33, 0.35};
```

The brackets for each rate for all the filing statuses can be represented in a two-dimensional array as follows:

```
int[][] brackets = {
  {8350, 33950, 82250, 171550, 372950},   // Single filer
  {16700, 67900, 137050, 20885, 372950}, // Married jointly
  {8350, 33950, 68525, 104425, 186475}, // Married separately
  {11950, 45500, 117450, 190200, 372950} // Head of household
};
```

Suppose the taxable income is $400,000 for single filers. The tax can be computed as follows:

```
tax = brackets[0][0] * rates[0] +
  (brackets[0][1] - brackets[0][0]) * rates[1] +
  (brackets[0][2] - brackets[0][1]) * rates[2] +
  (brackets[0][3] - brackets[0][2]) * rates[3] +
  (brackets[0][4] - brackets[0][3]) * rates[4] +
  (400000 - brackets[0][4]) * rates[5]
```

7.13* (*Locating the largest element*) Write the following method that returns the location of the largest element in a two-dimensional array.

```
public static int[] locateLargest(double[][] a)
```

The return value is a one-dimensional array that contains two elements. These two elements indicate the row and column indices of the largest element in the two-dimensional array. Write a test program that prompts the user to enter a two-dimensional array and displays the location of the largest element in the array. Here is a sample run:



```
Enter the number of rows and columns of the array: 3 4 Enter
Enter the array:
23.5 35 2 10 Enter
4.5 3 45 3.5 Enter
35 44 5.5 9.6 Enter
The location of the largest element is at (1, 2)
```

7.14** (*Exploring matrix*) Write a program that prompts the user to enter the length of a square matrix, randomly fills in 0s and 1s into the matrix, prints the matrix, and finds the rows, columns, and diagonals with all 0s or 1s. Here is a sample run of the program:



```
Enter the size for the matrix: 4 Enter
0111
0000
0100
1111
All 0s on row 1
All 1s on row 3
No same numbers on a column
No same numbers on the major diagonal
No same numbers on the sub-diagonal
```

7.15* (*Geometry: same line?*) Suppose a set of points are given. Write a program to check whether all the points are on the same line. Use the following sets to test your program:

```
double[][] set1 = {{1, 1}, {2, 2}, {3, 3}, {4, 4}};
double[][] set2 = {{0, 1}, {1, 2}, {4, 5}, {5, 6}};
double[][] set3 = {{0, 1}, {1, 2}, {4, 5}, {4.5, 4}};
```

7.16* (*Sorting two-dimensional array*) Write a method to sort a two-dimensional array using following header:

```
public static void sort(int m[][])
```

The method performs a primary sort on rows and a secondary sort on columns. For example, the array {{4, 2}, {1, 7}, {4, 5}, {1, 2}, {1, 1}, {4, 1}} will be sorted to {{1, 1}, {1, 2}, {1, 7}, {4, 1}, {4, 2}, {4, 5}}.

7.17* (*Financial tsunami*) Banks lend money to each other. In tough economic times, if a bank goes bankrupt, it may not be able to pay back the loan. A bank's total assets are its current balance plus its loans to other banks. Figure 7.8 is a diagram that shows five banks. The banks' current balances are 25, 125, 175, 75, and 181 million dollars, respectively. The directed edge from node 1 to node 2 indicates that bank 1 lends 40 million dollars to bank 2.

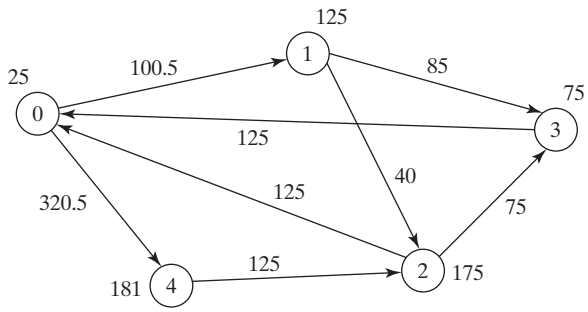


FIGURE 7.8 Banks lend money to each other.

If a bank's total assets are under a certain limit, the bank is unsafe. The money it borrowed cannot be returned to the lender, and the lender cannot count the loan in its total assets. Consequently, the lender may also be unsafe, if its total assets are under the limit. Write a program to find all unsafe banks. Your program reads the input as follows. It first reads two integers `n` and `limit`, where `n` indicates the number of banks and `limit` is the minimum total assets for keeping a bank safe. It then reads `n` lines that describe the information for `n` banks with id from `0` to `n-1`. The first number in the line is the bank's balance, the second number indicates the number of banks that borrowed money from the bank, and the rest are pairs of two numbers. Each pair describes a borrower. The first number in the pair is the borrower's id and the second is the amount borrowed. For example, the input for the five banks in Figure 7.8 is as follows (note that the limit is 201):

```
5 201
25 2 1 100.5 4 320.5
125 2 2 40 3 85
175 2 0 125 3 75
75 1 0 125
181 1 2 125
```

The total assets of bank 3 are (`75 + 125`), which is under `201`. So bank 3 is unsafe. After bank 3 becomes unsafe, the total assets of bank 1 fall below (`125 + 40`). So, bank 1 is also unsafe. The output of the program should be

Unsafe banks are 3 1

(*Hint*: Use a two-dimensional array `borrowers` to represent loans. `borrowers[i][j]` indicates the loan that bank i loans to bank j. Once bank j becomes unsafe, `borrowers[i][j]` should be set to `0`.)

7.18* (*Shuffling rows*) Write a method that shuffles the rows in a two-dimensional `int` array using the following header:

```
public static void shuffle(int[][] m)
```

Write a test program that shuffles the following matrix:

```
int[][] m = {{1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10}};
```

7.19** (*Pattern recognition: consecutive four equal numbers*) Write the following method that tests whether a two-dimensional array has four consecutive numbers of the same value, either horizontally, vertically, or diagonally.

```
public static boolean isConsecutiveFour(int[][] values)
```

Write a test program that prompts the user to enter the number of rows and columns of a two-dimensional array and then the values in the array and displays true if the array contains four consecutive numbers with the same value. Otherwise, display false. Here are some examples of the true cases:

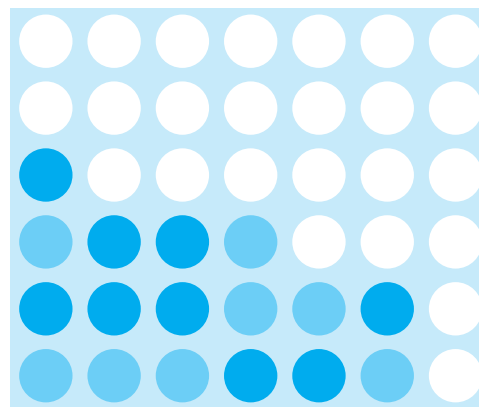
0	1	0	3	1	6	1
0	1	6	8	6	0	1
5	6	2	1	8	2	9
6	5	6	1	1	9	1
1	3	6	1	4	0	7
3	3	3	3	4	0	7

0	1	0	3	1	6	1
0	1	6	8	6	0	1
5	5	2	1	8	2	9
6	5	6	1	1	9	1
1	5	6	1	4	0	7
3	5	3	3	4	0	7

0	1	0	3	1	6	1
0	1	6	8	6	0	1
5	6	2	1	6	2	9
6	5	6	6	1	9	1
1	3	6	1	4	0	7
3	6	3	3	4	0	7

0	1	0	3	1	6	1
0	1	6	8	6	0	1
9	6	2	1	8	2	9
6	9	6	1	1	9	1
1	3	9	1	4	0	7
3	3	3	9	4	0	7

7.20** (*Game: connect four*) Connect four is a two-player board game in which the players alternately drop colored disks into a seven-column, six-row vertically-suspended grid, as shown below.



The objective of the game is to connect four same-colored disks in a row, a column, or a diagonal before your opponent can do likewise. The program prompts two players to drop a RED or YELLOW disk alternately. Whenever a disk is dropped, the program redisplays the board on the console and determines the status of the game (win, draw, or continue). Here is a sample run:



```
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
----------------------
Drop a red disk at column (0-6): 0 Enter
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
|R| | | | | | |
----------------------
Drop a yellow disk at column (0-6): 3 Enter
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
|R| | |Y| | | |
...
...
...
Drop a yellow disk at column (0-6): 6 Enter
| | | | | | | |
| | | | | | | |
| | | |R| | | |
| | | |Y|R|Y| |
| | |R|Y|Y|Y|Y|
|R|Y|R|Y|R|R|R|
----------------------
The yellow player won
```

7.21* (*Game: multiple Sudoku solutions*) The complete solution for the Sudoku problem is given in Supplement VII.A. A Sudoku problem may have multiple solutions. Modify Sudoku.java in Supplement VII.A to display the total number of the solutions. Display two solutions if multiple solutions exist.

7.22* (*Algebra: 2 × 2 matrix inverse*) The inverse of a square matrix `A` is denoted A^{-1}, such that $A \times A^{-1} = I$, where `I` is the identity matrix with all `1`'s on the diagonal and `0` on all other cells. For example, the inverse of matrix $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ is $\begin{bmatrix} -0.5 & 1 \\ 1.5 & 0 \end{bmatrix}$, i.e.,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} -0.5 & 1 \\ 1.5 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The inverse of a 2×2 matrix A can be obtained using the following formula:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \qquad A^{-1} = \frac{1}{ad - bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Implement the following method to obtain an inverse of the matrix:

```
public static double[][] inverse(double[][] A)
```

The method returns `null` if `ad - bc` is `0`.

Write a test program that prompts the user to enter `a`, `b`, `c`, `d` for a matrix and displays its inverse matrix. Here is a sample run:



```
Enter a, b, c, d: 1 2 3 4 Enter
-2.0 1.0
1.5 -0.5
```



```
Enter a, b, c, d: 0.5 2 1.5 4.5 Enter
-6.0 2.6666666666666665
2.0 -0.6666666666666666
```

7.23* (*Algebra:* 3×3 *matrix inverse*) The inverse of a square matrix `A` is denoted `A-1`, such that `A×A-1 = I`, where `I` is the identity matrix with all `1`'s on the diagonal and `0` on all other cells. The inverse of matrix $\begin{bmatrix} 1 & 2 & 1 \\ 2 & 3 & 1 \\ 4 & 5 & 3 \end{bmatrix}$, for example, is

$$\begin{bmatrix} -2 & 0.5 & 0.5 \\ 1 & 0.5 & -0.5 \\ 1 & -1.5 & 0.5 \end{bmatrix}$$

—that is,

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 3 & 1 \\ 4 & 5 & 3 \end{bmatrix} \times \begin{bmatrix} -2 & 0.5 & 0.5 \\ 1 & 0.5 & -0.5 \\ 1 & -1.5 & 0.5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The inverse of a 3×3 matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

can be obtained using the following formula if $|A| \neq 0$:

$$A^{-1} = \frac{1}{|A|}\begin{bmatrix} a_{22}a_{33} - a_{23}a_{32} & a_{13}a_{32} - a_{12}a_{33} & a_{12}a_{23} - a_{13}a_{22} \\ a_{23}a_{31} - a_{21}a_{33} & a_{11}a_{33} - a_{13}a_{31} & a_{13}a_{21} - a_{11}a_{23} \\ a_{21}a_{32} - a_{22}a_{31} & a_{12}a_{31} - a_{11}a_{32} & a_{11}a_{22} - a_{12}a_{21} \end{bmatrix}$$

$$|A| = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11}a_{22}a_{33} + a_{31}a_{12}a_{23} + a_{13}a_{21}a_{32}$$
$$- a_{13}a_{22}a_{31} - a_{11}a_{23}a_{32} - a_{33}a_{21}a_{12}.$$

Implement the following function to obtain an inverse of the matrix:

```
public static double[][] inverse(double[][] A)
```

The method returns `null` if `|A|` is `0`.

Write a test program that prompts the user to enter $a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33}$ for a matrix and displays its inverse matrix. Here is a sample run:

```
Enter a11, a12, a13, a21, a22, a23, a31, a32, a33:
  1 2 1 2 3 1 4 5 3  Enter
-2 0.5 0.5
1 0.5 -0.5
1 -1.5 0.5
```



```
Enter a11, a12, a13, a21, a22, a23, a31, a32, a33:
  1 4 2 2 5 8 2 1 8  Enter
2.0 -1.875 1.375
0.0 0.25 -0.25
-0.5 0.4375 -0.1875
```



This page intentionally left blank

CHAPTER 8

OBJECTS AND CLASSES

Objectives

- To describe objects and classes, and use classes to model objects (§8.2).
- To use UML graphical notations to describe classes and objects (§8.2).
- To demonstrate defining classes and creating objects (§8.3).
- To create objects using constructors (§8.4).
- To access objects via object reference variables (§8.5).
- To define a reference variable using a reference type (§8.5.1).
- To access an object's data and methods using the object member access operator (`.`) (§8.5.2).
- To define data fields of reference types and assign default values for an object's data fields (§8.5.3).
- To distinguish between object reference variables and primitive data type variables (§8.5.4).
- To use classes `Date`, `Random`, and `JFrame` in the Java library (§8.6).
- To distinguish between instance and static variables and methods (§8.7).
- To define private data fields with appropriate `get` and `set` methods (§8.8).
- To encapsulate data fields to make classes easy to maintain (§8.9).
- To develop methods with object arguments and differentiate between primitive-type arguments and object-type arguments (§8.10).
- To store and process objects in arrays (§8.11).



8.1 Introduction

Having learned the material in earlier chapters, you are able to solve many programming problems using selections, loops, methods, and arrays. However, these Java features are not sufficient for developing graphical user interfaces and large-scale software systems. Suppose you want to develop a GUI (graphical user interface, pronounced *goo-ee*) as shown in Figure 8.1. How do you program it?

why OOP?

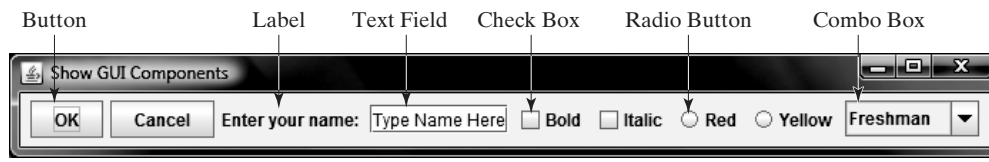


FIGURE 8.1 The GUI objects are created from classes.

This chapter begins the introduction of object-oriented programming, which will enable you to develop GUI and large-scale software systems effectively.

8.2 Defining Classes for Objects

object

Object-oriented programming (OOP) involves programming using objects. An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behavior.

state

- The *state* of an object (also known as its *properties* or *attributes*) is represented by *data fields* with their current values. A circle object, for example, has a data field `radius`, which is the property that characterizes a circle. A rectangle object has data fields `width` and `height`, which are the properties that characterize a rectangle.

behavior

- The *behavior* of an object (also known as its *actions*) is defined by methods. To invoke a method on an object is to ask the object to perform an action. For example, you may define a method named `getArea()` for circle objects. A circle object may invoke `getArea()` to return its area.

contract

instantiation

object

instance

Objects of the same type are defined using a common class. A class is a template, blueprint, or *contract* that defines what an object's data fields and methods will be. An object is an instance of a class. You can create many instances of a class. Creating an instance is referred to as *instantiation*. The terms *object* and *instance* are often interchangeable. The relationship between classes and objects is analogous to that between an apple-pie recipe and apple pies. You can make as many apple pies as you want from a single recipe. Figure 8.2 shows a class named `Circle` and its three objects.

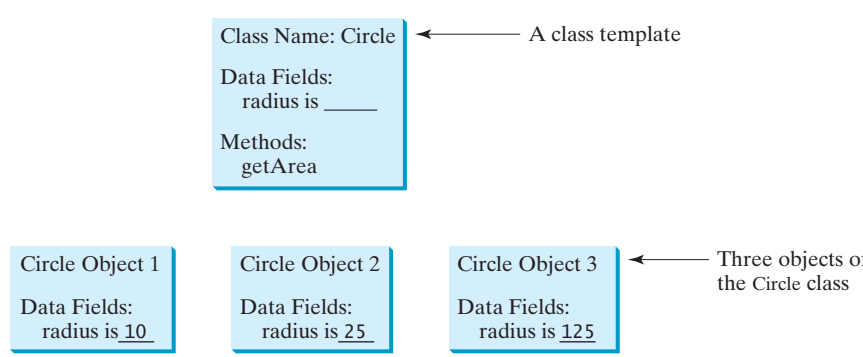


FIGURE 8.2 A class is a template for creating objects.

A Java class uses variables to define data fields and methods to define actions. Additionally, a class provides methods of a special type, known as *constructors*, which are invoked to create a new object. A constructor can perform any action, but constructors are designed to perform initializing actions, such as initializing the data fields of objects. Figure 8.3 shows an example of defining the class for circle objects.

class
data field
method
constructor

```
class Circle {
  /** The radius of this circle */
  double radius = 1.0;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * Math.PI;
  }
}
```

Data field

Constructors

Method

FIGURE 8.3 A class is a construct that defines objects of the same type.

The `Circle` class is different from all of the other classes you have seen thus far. It does not have a `main` method and therefore cannot be run; it is merely a definition for circle objects. The class that contains the `main` method will be referred to in this book, for convenience, as the *main class*.

main class

The illustration of class templates and objects in Figure 8.2 can be standardized using UML (Unified Modeling Language) notations. This notation, as shown in Figure 8.4, is called a *UML class diagram*, or simply a *class diagram*. In the class diagram, the data field is denoted as

class diagram

```
dataFieldName: dataFieldType
```

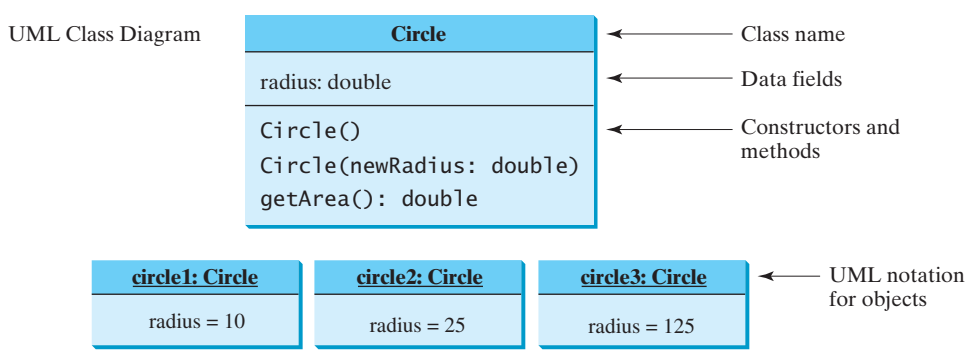


FIGURE 8.4 Classes and objects can be represented using UML notations.

The constructor is denoted as

```
ClassName(parameterName: parameterType)
```

The method is denoted as

```
methodName(parameterName: parameterType): returnType
```

8.3 Example: Defining Classes and Creating Objects

This section gives two examples of defining classes and uses the classes to create objects. Listing 8.1 is a program that defines the `Circle` class and uses it to create objects. To avoid a naming conflict with several improved versions of the `Circle` class introduced later in this book, the `Circle` class in this example is named `Circle1`.

The program constructs three circle objects with radius `1.0`, `25`, and `125` and displays the radius and area of each of the three circles. Change the radius of the second object to `100` and display its new radius and area.

LISTING 8.1 TestCircle1.java

main class
main method
create object
create object
create object
class `Circle1`
data field
no-arg constructor
second constructor
method

```
1  public class TestCircle1 {
2    /** Main method */
3    public static void main(String[] args) {
4      // Create a circle with radius 1.0
5      Circle1 circle1 = new Circle1();
6      System.out.println("The area of the circle of radius "
7        + circle1.radius + " is " + circle1.getArea());
8
9      // Create a circle with radius 25
10     Circle1 circle2 = new Circle1(25);
11     System.out.println("The area of the circle of radius "
12       + circle2.radius + " is " + circle2.getArea());
13
14     // Create a circle with radius 125
15     Circle1 circle3 = new Circle1(125);
16     System.out.println("The area of the circle of radius "
17       + circle3.radius + " is " + circle3.getArea());
18
19     // Modify circle radius
20     circle2.radius = 100;
21     System.out.println("The area of the circle of radius "
22       + circle2.radius + " is " + circle2.getArea());
23   }
24 }
25
26 // Define the circle class with two constructors
27 class Circle1 {
28   double radius;
29
30   /** Construct a circle with radius 1 */
31   Circle1() {
32     radius = 1.0;
33   }
34
35   /** Construct a circle with a specified radius */
36   Circle1(double newRadius) {
37     radius = newRadius;
38   }
39
40   /** Return the area of this circle */
41   double getArea() {
42     return radius * radius * Math.PI;
43   }
44 }
```

```
The area of the circle of radius 1.0 is 3.141592653589793
The area of the circle of radius 25.0 is 1963.4954084936207
The area of the circle of radius 125.0 is 49087.385212340516
The area of the circle of radius 100.0 is 31415.926535897932
```



The program contains two classes. The first of these, `TestCircle1`, is the main class. Its sole purpose is to test the second class, `Circle1`. Such a program that uses the class is often referred to as a *client* of the class. When you run the program, the Java runtime system invokes the `main` method in the main class.

client

You can put the two classes into one file, but only one class in the file can be a public class. Furthermore, the public class must have the same name as the file name. Therefore, the file name is TestCircle1.java, since `TestCircle1` is public.

public class

The main class contains the `main` method (line 3) that creates three objects. As in creating an array, the `new` operator is used to create an object from the constructor. `new Circle1()` creates an object with radius `1.0` (line 5), `new Circle1(25)` creates an object with radius `25` (line 10), and `new Circle1(125)` creates an object with radius `125` (line 15).

These three objects (referenced by `circle1`, `circle2`, and `circle3`) have different data but the same methods. Therefore, you can compute their respective areas by using the `getArea()` method. The data fields can be accessed via the reference of the object using `circle1.radius`, `circle2.radius`, and `circle3.radius`, respectively. The object can invoke its method via the reference of the object using `circle1.getArea()`, `circle2.getArea()`, and `circle3.getArea()`, respectively.

These three objects are independent. The radius of `circle2` is changed to `100` in line 20. The object's new radius and area is displayed in lines 21–22.

There are many ways to write Java programs. For instance, you can combine the two classes in the example into one, as shown in Listing 8.2.

LISTING 8.2 Circle1.java

```
1  public class Circle1 {
2    /** Main method */
3    public static void main(String[] args) {
4      // Create a circle with radius 1.0
5      Circle1 circle1 = new Circle1();
6      System.out.println("The area of the circle of radius "
7        + circle1.radius + " is " + circle1.getArea());
8
9      // Create a circle with radius 25
10     Circle1 circle2 = new Circle1(25);
11     System.out.println("The area of the circle of radius "
12       + circle2.radius + " is " + circle2.getArea());
13
14     // Create a circle with radius 125
15     Circle1 circle3 = new Circle1(125);
16     System.out.println("The area of the circle of radius "
17       + circle3.radius + " is " + circle3.getArea());
18
19     // Modify circle radius
20     circle2.radius = 100;
21     System.out.println("The area of the circle of radius "
22       + circle2.radius + " is " + circle2.getArea());
23   }
24
```

main method

data field

no-arg constructor

second constructor

method

```
25   double radius;
26
27   /** Construct a circle with radius 1 */
28   Circle1() {
29     radius = 1.0;
30   }
31
32   /** Construct a circle with a specified radius */
33   Circle1(double newRadius) {
34     radius = newRadius;
35   }
36
37   /** Return the area of this circle */
38   double getArea() {
39     return radius * radius * Math.PI;
40   }
41 }
```

Since the combined class has a `main` method, it can be executed by the Java interpreter. The `main` method is the same as in Listing 1.1. This demonstrates that you can test a class by simply adding a `main` method in the same class.

As another example, consider TV sets. Each TV is an object with states (current channel, current volume level, power on or off) and behaviors (change channels, adjust volume, turn on/off). You can use a class to model TV sets. The UML diagram for the class is shown in Figure 8.5.

Listing 8.3 gives a program that defines the `TV` class.

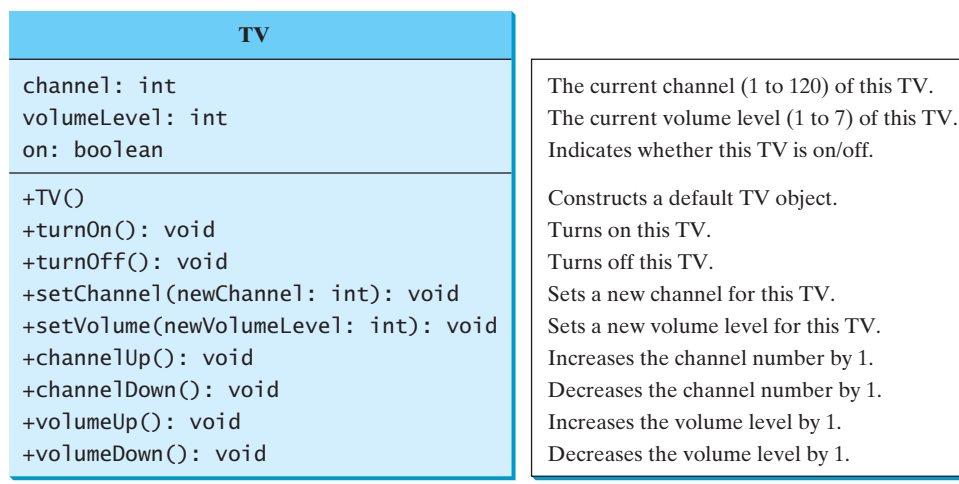

TV
channel: int
volumeLevel: int
on: boolean
+TV()
+turnOn(): void
+turnOff(): void
+setChannel(newChannel: int): void
+setVolume(newVolumeLevel: int): void
+channelUp(): void
+channelDown(): void
+volumeUp(): void
+volumeDown(): void
The current channel (1 to 120) of this TV.
The current volume level (1 to 7) of this TV.
Indicates whether this TV is on/off.
Constructs a default TV object.
Turns on this TV.
Turns off this TV.
Sets a new channel for this TV.
Sets a new volume level for this TV.
Increases the channel number by 1.
Decreases the channel number by 1.
Increases the volume level by 1.
Decreases the volume level by 1.

FIGURE 8.5 The TV class models TV sets.

LISTING 8.3 TV.java

data fields

constructor

```
1 public class TV {
2   int channel = 1; // Default channel is 1
3   int volumeLevel = 1; // Default volume level is 1
4   boolean on = false; // By default TV is off
5
6   public TV() {
7   }
8
```

```
 9   public void turnOn() {                                    turn on TV
10     on = true;
11   }
12
13   public void turnOff() {                                   turn off TV
14     on = false;
15   }
16
17   public void setChannel(int newChannel) {                  set a new channel
18     if (on && newChannel >= 1 && newChannel <= 120)
19       channel = newChannel;
20   }
21
22   public void setVolume(int newVolumeLevel) {               set a new volume
23     if (on && newVolumeLevel >= 1 && newVolumeLevel <= 7)
24       volumeLevel = newVolumeLevel;
25   }
26
27   public void channelUp() {                                 increase channel
28     if (on && channel < 120)
29       channel++;
30   }
31
32   public void channelDown() {                               decrease channel
33     if (on && channel > 1)
34       channel--;
35   }
36
37   public void volumeUp() {                                  increase volume
38     if (on && volumeLevel < 7)
39       volumeLevel++;
40   }
41
42   public void volumeDown() {                                decrease volume
43     if (on && volumeLevel > 1)
44       volumeLevel--;
45   }
46 }
```

Note that the channel and volume level are not changed if the TV is not on. Before either of these is changed, its current value is checked to ensure that it is within the correct range.

Listing 8.4 gives a program that uses the `TV` class to create two objects.

LISTING 8.4 TestTV.java

```
 1 public class TestTV {
 2   public static void main(String[] args) {                  main method
 3     TV tv1 = new TV();                                      create a TV
 4     tv1.turnOn();                                           turn on
 5     tv1.setChannel(30);                                     set a new channel
 6     tv1.setVolume(3);                                       set a new volume
 7
 8     TV tv2 = new TV();                                      create a TV
 9     tv2.turnOn();                                           turn on
10     tv2.channelUp();                                        increase channel
11     tv2.channelUp();
12     tv2.volumeUp();                                         increase volume
13
14     System.out.println("tv1's channel is " + tv1.channel   display state
15       + " and volume level is " + tv1.volumeLevel );
```

```
16     System.out.println("tv2's channel is " + tv2.channel
17       + " and volume level is " + tv2.volumeLevel);
18   }
19 }
```



```
tv1's channel is 30 and volume level is 3
tv2's channel is 3 and volume level is 2
```

The program creates two objects in lines 3 and 8 and invokes the methods on the objects to perform actions for setting channels and volume levels and for increasing channels and volumes. The program displays the state of the objects in lines 14–17. The methods are invoked using a syntax such as `tv1.turnOn()` (line 4). The data fields are accessed using a syntax such as `tv1.channel` (line 14).

These examples have given you a glimpse of classes and objects. You may have many questions regarding constructors, objects, reference variables, and accessing data fields, and invoking object's methods. The sections that follow discuss these issues in detail.

8.4 Constructing Objects Using Constructors

Constructors are a special kind of method. They have three peculiarities:

constructor's name

- A constructor must have the same name as the class itself.

no return type

- Constructors do not have a return type—not even `void`.

new operator

- Constructors are invoked using the `new` operator when an object is created. Constructors play the role of initializing objects.

overloaded constructors

The constructor has exactly the same name as the defining class. Like regular methods, constructors can be overloaded (i.e., multiple constructors can have the same name but different signatures), making it easy to construct objects with different initial data values.

no **void**

It is a common mistake to put the `void` keyword in front of a constructor. For example,

```
public void Circle() {
}
```

In this case, `Circle()` is a method, not a constructor.

constructing objects

Constructors are used to construct objects. To construct an object from a class, invoke a constructor of the class using the `new` operator, as follows:

```
new ClassName(arguments);
```

For example, `new Circle()` creates an object of the `Circle` class using the first constructor defined in the `Circle` class, and `new Circle(25)` creates an object using the second constructor defined in the `Circle` class.

A class normally provides a constructor without arguments (e.g., `Circle()`). Such a constructor is referred to as a *no-arg* or *no-argument constructor*.

no-arg constructor

default constructor

A class may be defined without constructors. In this case, a no-arg constructor with an empty body is implicitly defined in the class. This constructor, called *a default constructor,* is provided automatically *only if no constructors are explicitly defined in the class.*

8.5 Accessing Objects via Reference Variables

Newly created objects are allocated in the memory. They can be accessed via reference variables.

8.5.1 Reference Variables and Reference Types

reference variable

Objects are accessed via object *reference variables*, which contain references to the objects. Such variables are declared using the following syntax:

```
ClassName objectRefVar;
```

reference type

A class is essentially a programmer-defined type. A class is a *reference type*, which means that a variable of the class type can reference an instance of the class. The following statement declares the variable `myCircle` to be of the `Circle` type:

```
Circle myCircle;
```

The variable `myCircle` can reference a `Circle` object. The next statement creates an object and assigns its reference to `myCircle`:

```
myCircle = new Circle();
```

Using the syntax shown below, you can write a single statement that combines the declaration of an object reference variable, the creation of an object, and the assigning of an object reference to the variable.

```
ClassName objectRefVar = new ClassName();
```

Here is an example:

```
Circle myCircle = new Circle();
```

The variable `myCircle` holds a reference to a `Circle` object.



Note

object vs. object reference variable

An object reference variable that appears to hold an object actually contains a reference to that object. Strictly speaking, an object reference variable and an object are different, but most of the time the distinction can be ignored. So it is fine, for simplicity, to say that `myCircle` is a `Circle` object rather than use the longer-winded description that `myCircle` is a variable that contains a reference to a `Circle` object.



Note

array object

Arrays are treated as objects in Java. Arrays are created using the `new` operator. An array variable is actually a variable that contains a reference to an array.

8.5.2 Accessing an Object's Data and Methods

dot operator

After an object is created, its data can be accessed and its methods invoked using the dot operator (`.`), also known as the *object member access operator*:

- `objectRefVar.dataField` references a data field in the object.
- `objectRefVar.method(arguments)` invokes a method on the object.

For example, `myCircle.radius` references the radius in `myCircle`, and `myCircle.getArea()` invokes the `getArea` method on `myCircle`. Methods are invoked as operations on objects.

instance variable
instance method

The data field `radius` is referred to as an *instance variable*, because it is dependent on a specific instance. For the same reason, the method `getArea` is referred to as an *instance method*, because you can invoke it only on a specific instance. The object on which an instance method is invoked is called a *calling object*.

calling object



Caution

invoking methods

Recall that you use `Math.methodName(arguments)` (e.g., `Math.pow(3, 2.5)`) to invoke a method in the `Math` class. Can you invoke `getArea()` using `Circle.getArea()`? The answer is no. All the methods in the `Math` class are static methods, which are defined using the `static` keyword. However, `getArea()` is an instance method, and thus nonstatic. It must be invoked from an object using `objectRefVar.methodName(arguments)` (e.g., `myCircle.getArea()`). Further explanation is given in §8.7, "Static Variables, Constants, and Methods."



Note

Usually you create an object and assign it to a variable. Later you can use the variable to reference the object. Occasionally an object does not need to be referenced later. In this case, you can create an object without explicitly assigning it to a variable, as shown below:

```
new Circle();
```

or

```
System.out.println("Area is " + new Circle(5).getArea());
```

anonymous object

The former statement creates a `Circle` object. The latter creates a `Circle` object and invokes its `getArea` method to return its area. An object created in this way is known as an *anonymous object*.

8.5.3 Reference Data Fields and the `null` Value

reference data fields

The data fields can be of reference types. For example, the following `Student` class contains a data field `name` of the `String` type. `String` is a predefined Java class.

```
class Student {
  String name; // name has default value null
  int age; // age has default value 0
  boolean isScienceMajor; // isScienceMajor has default value false
  char gender; // c has default value '\u0000'
}
```

`null` value

If a data field of a reference type does not reference any object, the data field holds a special Java value, `null`. `null` is a literal just like `true` and `false`. While `true` and `false` are Boolean literals, `null` is a literal for a reference type.

default field values

The default value of a data field is `null` for a reference type, `0` for a numeric type, `false` for a `boolean` type, and `'\u0000'` for a `char` type. However, Java assigns no default value to a local variable inside a method. The following code displays the default values of data fields `name`, `age`, `isScienceMajor`, and `gender` for a `Student` object:

```
class Test {
  public static void main(String[] args) {
    Student student = new Student();
    System.out.println("name? " + student.name);
    System.out.println("age? " + student.age);
    System.out.println("isScienceMajor? " + student.isScienceMajor);
    System.out.println("gender? " + student.gender);
  }
}
```

The code below has a compile error, because local variables `x` and `y` are not initialized:

```
class Test {
  public static void main(String[] args) {
```

```
    int x; // x has no default value
    String y; // y has no default value
    System.out.println("x is " + x);
    System.out.println("y is " + y);
  }
}
```



Caution

`NullPointerException` is a common runtime error. It occurs when you invoke a method on a reference variable with `null` value. Make sure you assign an object reference to the variable before invoking the method through the reference variable.

`NullPointerException`

8.5.4 Differences Between Variables of Primitive Types and Reference Types

Every variable represents a memory location that holds a value. When you declare a variable, you are telling the compiler what type of value the variable can hold. For a variable of a primitive type, the value is of the primitive type. For a variable of a reference type, the value is a reference to where an object is located. For example, as shown in Figure 8.6, the value of `int` variable `i` is `int` value `1`, and the value of `Circle` object `c` holds a reference to where the contents of the `Circle` object are stored in the memory.

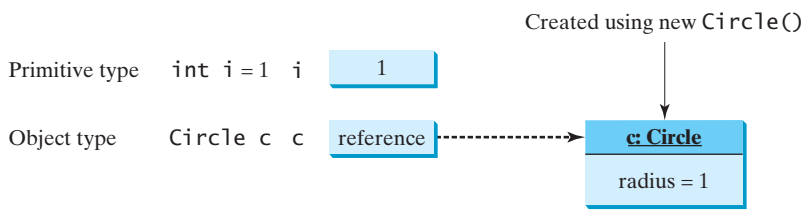


FIGURE 8.6 A variable of a primitive type holds a value of the primitive type, and a variable of a reference type holds a reference to where an object is stored in memory.

When you assign one variable to another, the other variable is set to the same value. For a variable of a primitive type, the real value of one variable is assigned to the other variable. For a variable of a reference type, the reference of one variable is assigned to the other variable. As shown in Figure 8.7, the assignment statement `i = j` copies the contents of `j` into `i` for primitive variables. As shown in Figure 8.8, the assignment statement `c1 = c2` copies the reference of `c2` into `c1` for reference variables. After the assignment, variables `c1` and `c2` refer to the same object.

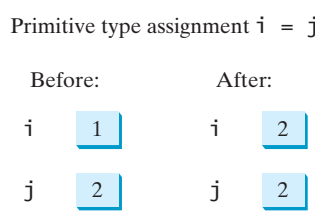


FIGURE 8.7 Primitive variable `j` is copied to variable `i`.

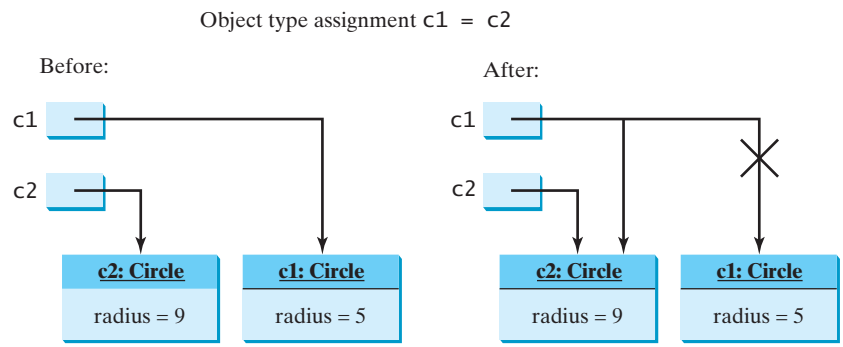


FIGURE 8.8 Reference variable `c2` is copied to variable `c1`.



Note

As shown in Figure 8.8, after the assignment statement `c1 = c2`, `c1` points to the same object referenced by `c2`. The object previously referenced by `c1` is no longer useful and therefore is now known as *garbage*. Garbage occupies memory space. The Java runtime system detects garbage and automatically reclaims the space it occupies. This process is called *garbage collection*.

garbage
garbage collection



Tip

If you know that an object is no longer needed, you can explicitly assign `null` to a reference variable for the object. The JVM will automatically collect the space if the object is not referenced by any reference variable.

8.6 Using Classes from the Java Library

Listing 8.1 defined the `Circle1` class and created objects from the class. You will frequently use the classes in the Java library to develop programs. This section gives some examples of the classes in the Java library.

8.6.1 The `Date` Class

In Listing 2.8, ShowCurrentTime.java, you learned how to obtain the current time using `System.currentTimeMillis()`. You used the division and remainder operators to extract current second, minute, and hour. Java provides a system-independent encapsulation of date and time in the `java.util.Date` class, as shown in Figure 8.9.

`java.util.Date` class

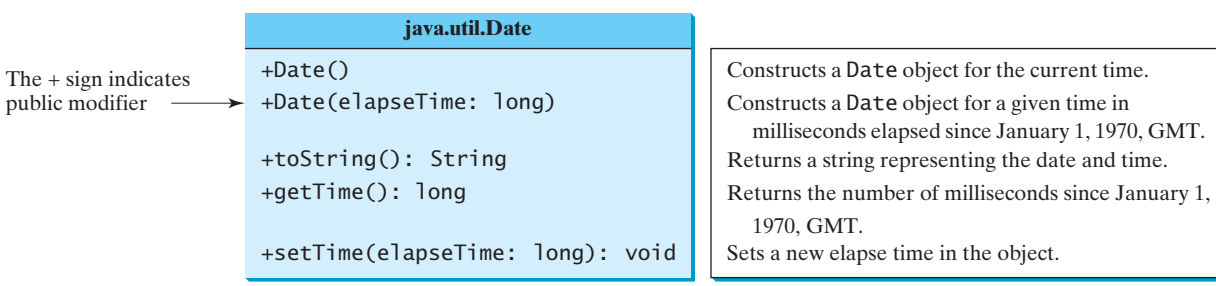

The + sign indicates public modifier
java.util.Date
+Date()
+Date(elapseTime: long)
+toString(): String
+getTime(): long
+setTime(elapseTime: long): void
Constructs a Date object for the current time.
Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT.
Returns a string representing the date and time.
Returns the number of milliseconds since January 1, 1970, GMT.
Sets a new elapse time in the object.

FIGURE 8.9 A `Date` object represents a specific date and time.

You can use the no-arg constructor in the `Date` class to create an instance for the current date and time, its `getTime()` method to return the elapsed time since January 1, 1970, GMT, and its `toString` method to return the date and time as a string. For example, the following code

```
java.util.Date date = new java.util.Date();
System.out.println("The elapsed time since Jan 1, 1970 is " +
  date.getTime() + " milliseconds");
System.out.println(date.toString());
```

create object

get elapsed time
invoke **toString**

displays the output like this:

```
The elapsed time since Jan 1, 1970 is 1100547210284 milliseconds
Mon Nov 15 14:33:30 EST 2004
```

The `Date` class has another constructor, `Date(long elapseTime)`, which can be used to construct a `Date` object for a given time in milliseconds elapsed since January 1, 1970, GMT.

8.6.2 The `Random` Class

You have used `Math.random()` to obtain a random `double` value between `0.0` and `1.0` (excluding `1.0`). Another way to generate random numbers is to use the `java.util.Random` class, as shown in Figure 8.10, which can generate a random `int`, `long`, `double`, `float`, and `boolean` value.

java.util.Random	
+Random()	Constructs a Random object with the current time as its seed.
+Random(seed: long)	Constructs a Random object with a specified seed.
+nextInt(): int	Returns a random int value.
+nextInt(n: int): int	Returns a random int value between 0 and n (exclusive).
+nextLong(): long	Returns a random long value.
+nextDouble(): double	Returns a random double value between 0.0 and 1.0 (exclusive).
+nextFloat(): float	Returns a random float value between 0.0F and 1.0F (exclusive).
+nextBoolean(): boolean	Returns a random boolean value.

FIGURE 8.10 A `Random` object can be used to generate random values.

When you create a `Random` object, you have to specify a seed or use the default seed. The no-arg constructor creates a `Random` object using the current elapsed time as its seed. If two `Random` objects have the same seed, they will generate identical sequences of numbers. For example, the following code creates two `Random` objects with the same seed, `3`.

```
Random random1 = new Random(3);
System.out.print("From random1: ");
for (int i = 0; i < 10; i++)
  System.out.print(random1.nextInt(1000) + " ");

Random random2 = new Random(3);
System.out.print("\nFrom random2: ");
for (int i = 0; i < 10; i++)
  System.out.print(random2.nextInt(1000) + " ");
```

The code generates the same sequence of random `int` values:

```
From random1: 734 660 210 581 128 202 549 564 459 961
From random2: 734 660 210 581 128 202 549 564 459 961
```



Note

The ability to generate the same sequence of random values is useful in software testing and many other applications. In software testing, you can test your program using a fixed sequence of numbers before using different sequences of random numbers.

same sequence

8.6.3 Displaying GUI Components



Pedagogical Note

Graphical user interface (GUI) components are good examples for teaching OOP. Simple GUI examples are introduced for this purpose. The full introduction to GUI programming begins with Chapter 12, "GUI Basics."

When you develop programs to create graphical user interfaces, you will use Java classes such as `JFrame`, `JButton`, `JRadioButton`, `JComboBox`, and `JList` to create frames, buttons, radio buttons, combo boxes, lists, and so on. Listing 8.5 is an example that creates two windows using the `JFrame` class. The output of the program is shown in Figure 8.11.



FIGURE 8.11 The program creates two windows using the `JFrame` class.

LISTING 8.5 `TestFrame.java`

```
1 import javax.swing.JFrame;
2
3 public class TestFrame {
4   public static void main(String[] args) {
5     JFrame frame1 = new JFrame();
6     frame1.setTitle("Window 1");
7     frame1.setSize(200, 150);
8     frame1.setLocation(200, 100);
9     frame1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10    frame1.setVisible(true);
11
12    JFrame frame2 = new JFrame();
13    frame2.setTitle("Window 2");
14    frame2.setSize(200, 150);
15    frame2.setLocation(410, 100);
16    frame2.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17    frame2.setVisible(true);
18  }
19 }
```

create an object
invoke a method

create an object
invoke a method

This program creates two objects of the `JFrame` class (lines 5, 12) and then uses the methods `setTitle`, `setSize`, `setLocation`, `setDefaultCloseOperation`, and `setVisible` to set the properties of the objects. The `setTitle` method sets a title for the window (lines 6, 13). The `setSize` method sets the window's width and height (lines 7, 14). The `setLocation` method specifies the location of the window's upper-left corner (lines 8, 15). The `setDefaultCloseOperation` method terminates the program when the frame is closed (lines 9, 16). The `setVisible` method displays the window.

You can add graphical user interface components, such as buttons, labels, text fields, check boxes, and combo boxes to the window. The components are defined using classes. Listing 8.6 gives an example of creating a graphical user interface, as shown in Figure 8.1.

LISTING 8.6 GUIComponents.java



Video Note
Use classes

```
1 import javax.swing.*;
2
3 public class GUIComponents {
4   public static void main(String[] args) {
5     // Create a button with text OK
6     JButton jbtOK = new JButton("OK");                          create a button
7
8     // Create a button with text Cancel
9     JButton jbtCancel = new JButton("Cancel");                  create a button
10
11    // Create a label with text "Enter your name: "
12    JLabel jlblName = new JLabel("Enter your name: ");          create a label
13
14    // Create a text field with text "Type Name Here"
15    JTextField jtfName = new JTextField("Type Name Here");      create a text field
16
17    // Create a check box with text bold
18    JCheckBox jchkBold = new JCheckBox("Bold");                 create a check box
19
20    // Create a check box with text italic
21    JCheckBox jchkItalic = new JCheckBox("Italic");             create a check box
22
23    // Create a radio button with text red
24    JRadioButton jrbRed = new JRadioButton("Red");              create a radio button
25
26    // Create a radio button with text yellow
27    JRadioButton jrbYellow = new JRadioButton("Yellow");        create a radio button
28
29    // Create a combo box with several choices
30    JComboBox jcboColor = new JComboBox(new String[]{"Freshman",  create a combo box
31      "Sophomore", "Junior", "Senior"});
32
33    // Create a panel to group components
34    JPanel panel = new JPanel();                                create a panel
35    panel.add(jbtOK); // Add the OK button to the panel         add to panel
36    panel.add(jbtCancel); // Add the Cancel button to the panel
37    panel.add(jlblName); // Add the label to the panel
38    panel.add(jtfName); // Add the text field to the panel
39    panel.add(jchkBold); // Add the check box to the panel
40    panel.add(jchkItalic); // Add the check box to the panel
41    panel.add(jrbRed); // Add the radio button to the panel
42    panel.add(jrbYellow); // Add the radio button to the panel
43    panel.add(jcboColor); // Add the combo box to the panel
44
45    JFrame frame = new JFrame(); // Create a frame              create a frame
46    frame.add(panel); // Add the panel to the frame            add panel to frame
47    frame.setTitle("Show GUI Components");
48    frame.setSize(450, 100);
49    frame.setLocation(200, 100);
50    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
51    frame.setVisible(true);                                     display frame
52  }
53 }
```

This program creates GUI objects using the classes `JButton`, `JLabel`, `JTextField`, `JCheckBox`, `JRadioButton`, and `JComboBox` (lines 6–31). Then, using the `JPanel` class (line 34), it then creates a panel object and adds to it the button, label, text field, check box,

radio button, and combo box (lines 35–43). The program then creates a frame and adds the panel to the frame (line 45). The frame is displayed in line 51.

8.7 Static Variables, Constants, and Methods

instance variable

The data field `radius` in the circle class in Listing 8.1 is known as an *instance variable*. An instance variable is tied to a specific instance of the class; it is not shared among objects of the same class. For example, suppose that you create the following objects:



Video Note
static vs. instance

```
Circle circle1 = new Circle();
Circle circle2 = new Circle(5);
```

The `radius` in `circle1` is independent of the `radius` in `circle2` and is stored in a different memory location. Changes made to `circle1`'s `radius` do not affect `circle2`'s `radius`, and vice versa.

static variable

If you want all the instances of a class to share data, use *static variables*, also known as *class variables*. Static variables store values for the variables in a common memory location. Because of this common location, if one object changes the value of a static variable, all objects of the same class are affected. Java supports static methods as well as static variables.

static method

Static methods can be called without creating an instance of the class.

Let us modify the `Circle` class by adding a static variable `numberOfObjects` to count the number of circle objects created. When the first object of this class is created, `numberOfObjects` is `1`. When the second object is created, `numberOfObjects` becomes `2`. The UML of the new circle class is shown in Figure 8.12. The `Circle` class defines the instance variable `radius` and the static variable `numberOfObjects`, the instance methods `getRadius`, `setRadius`, and `getArea`, and the static method `getNumberOfObjects`. (Note that static variables and methods are underlined in the UML class diagram.)

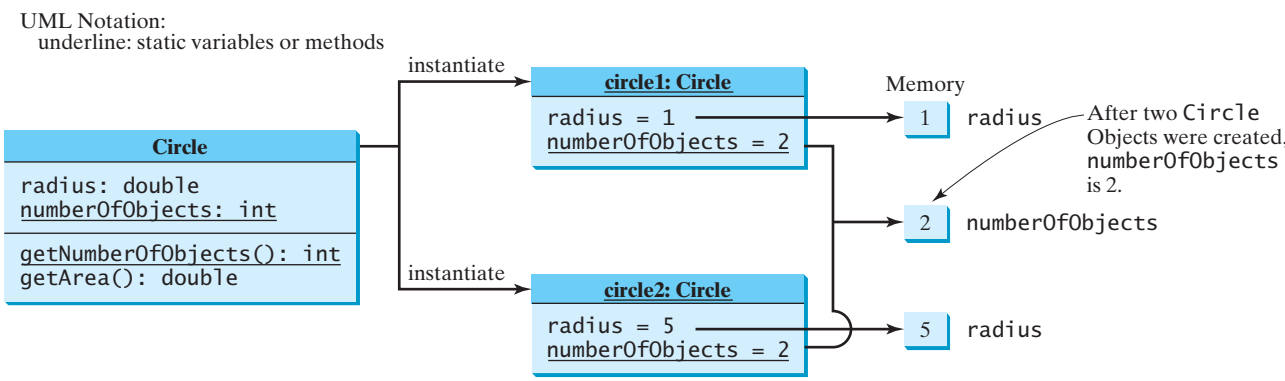


FIGURE 8.12 Instance variables belong to the instances and have memory storage independent of one another. Static variables are shared by all the instances of the same class.

To declare a static variable or define a static method, put the modifier `static` in the variable or method declaration. The static variable `numberOfObjects` and the static method `getNumberOfObjects()` can be declared as follows:

declare static variable

```
static int numberOfObjects;
```

define static method

```
static int getNumberObjects() {
  return  numberOfObjects;
}
```

Constants in a class are shared by all objects of the class. Thus, constants should be declared `final static`. For example, the constant `PI` in the `Math` class is defined as:

declare constant

```
final static double PI = 3.14159265358979323846;
```

The new circle class, named **Circle2**, is declared in Listing 8.7:

LISTING 8.7 Circle2.java

```
1 public class Circle2 {
2   /** The radius of the circle */
3   double radius;
4
5   /** The number of objects created */
6   static int numberOfObjects = 0;                          static variable
7
8   /** Construct a circle with radius 1 */
9   Circle2() {
10    radius = 1.0;
11    numberOfObjects++;                                     increase by 1
12  }
13
14  /** Construct a circle with a specified radius */
15  Circle2(double newRadius) {
16    radius = newRadius;
17    numberOfObjects++;                                     increase by 1
18  }
19
20  /** Return numberOfObjects */
21  static int getNumberOfObjects() {                        static method
22    return numberOfObjects;
23  }
24
25  /** Return the area of this circle */
26  double getArea() {
27    return radius * radius * Math.PI;
28  }
29 }
```

Method **getNumberOfObjects()** in **Circle2** is a static method. Other examples of static methods are **showMessageDialog** and **showInputDialog** in the **JOptionPane** class and all the methods in the **Math** class. The **main** method is static, too.

Instance methods (e.g., **getArea()**) and instance data (e.g., **radius**) belong to instances and can be used only after the instances are created. They are accessed via a reference variable. Static methods (e.g., **getNumberOfObjects()**) and static data (e.g., **numberOfObjects**) can be accessed from a reference variable or from their class name.

The program in Listing 8.8 demonstrates how to use instance and static variables and methods and illustrates the effects of using them.

LISTING 8.8 TestCircle2.java

```
1 public class TestCircle2 {
2   /** Main method */
3   public static void main(String[] args) {
4     System.out.println("Before creating objects");
5     System.out.println("The number of Circle objects is " +
6       Circle2.numberOfObjects);                            static variable
7
8     // Create c1
9     Circle2 c1 = new Circle2();
10
11    // Display c1 BEFORE c2 is created
12    System.out.println("\nAfter creating c1");
13    System.out.println("c1: radius (" + c1.radius +       instance variable
14      ") and number of Circle objects (" +
```

static variable

```
15        c1.numberOfObjects + ")");
16
17      // Create c2
18      Circle2 c2 = new Circle2(5);
19
20      // Modify c1
```

instance variable

```
21      c1.radius = 9;
22
23      // Display c1 and c2 AFTER c2 was created
24      System.out.println("\nAfter creating c2 and modifying c1");
25      System.out.println("c1: radius (" + c1.radius +
26        ") and number of Circle objects (" +
```

static variable

```
27        c1.numberOfObjects + ")");
28      System.out.println("c2: radius (" + c2.radius +
29        ") and number of Circle objects (" +
```

static variable

```
30        c2.numberOfObjects + ")");
31    }
32 }
```



```
Before creating objects
The number of Circle objects is 0

After creating c1
c1: radius (1.0) and number of Circle objects (1)

After creating c2 and modifying c1
c1: radius (9.0) and number of Circle objects (2)
c2: radius (5.0) and number of Circle objects (2)
```

When you compile **TestCircle2.java**, the Java compiler automatically compiles **Circle2.java** if it has not been compiled since the last change.

Static variables and methods can be accessed without creating objects. Line 6 displays the number of objects, which is **0**, since no objects have been created.

The **main** method creates two circles, **c1** and **c2** (lines 9, 18). The instance variable **radius** in **c1** is modified to become **9** (line 21). This change does not affect the instance variable **radius** in **c2**, since these two instance variables are independent. The static variable **numberOfObjects** becomes **1** after **c1** is created (line 9), and it becomes **2** after **c2** is created (line 18).

Note that **PI** is a constant defined in **Math**, and **Math.PI** references the constant. **c.numberOfObjects** could be replaced by **Circle2.numberOfObjects**. This improves readability, because the reader can easily recognize the static variable. You can also replace **Circle2.numberOfObjects** by **Circle2.getNumberOfObjects()**.



Tip

use class name

Use **ClassName.methodName(arguments)** to invoke a static method and **ClassName.staticVariable** to access a static variable. This improves readability, because the user can easily recognize the static method and data in the class.

Static variables and methods can be used from instance or static methods in the class. However, instance variables and methods can be used only from instance methods, not from static methods, since static variables and methods don't belong to a particular object. Thus the code given below is wrong.

```
1 public class Foo {
2   int i = 5;
3   static int k = 2;
4
5   public static void main(String[] args) {
6     int j = i; // Wrong because i is an instance variable
7     m1(); // Wrong because m1() is an instance method
```

```
 8   }
 9
10   public void m1() {
11     // Correct since instance and static variables and methods
12     // can be used in an instance method
13     i = i + k + m2(i, k);
14   }
15
16   public static int m2(int i, int j) {
17     return (int)(Math.pow(i, j));
18   }
19 }
```

Note that if you replace the code in lines 5–8 with the following new code, the program is fine, because the instance data field `i` and method `m1` are now accessed from an object `foo` (lines 6–7):

```
 1 public class Foo {
 2   int i = 5;
 3   static int k = 2;
 4
 5   public static void main(String[] args) {
 6     Foo foo = new Foo();
 7     int j = foo.i; // OK, foo.i accesses the object's instance variable
 8     foo.m1();   // OK. Foo.m1() invokes object's instance method
 9   }
10
11   public void m1() {
12     i = i + k + m2(i, k);
13   }
14
15   public static int m2(int i, int j) {
16     return (int)(Math.pow(i, j));
17   }
18 }
```



Design Guide

instance or static?

How do you decide whether a variable or method should be an instance one or a static one? A variable or method that is dependent on a specific instance of the class should be an instance variable or method. A variable or method that is not dependent on a specific instance of the class should be a static variable or method. For example, every circle has its own radius. Radius is dependent on a specific circle. Therefore, `radius` is an instance variable of the `Circle` class. Since the `getArea` method is dependent on a specific circle, it is an instance method. None of the methods in the `Math` class, such as `random`, `pow`, `sin`, and `cos`, is dependent on a specific instance. Therefore, these methods are static methods. The `main` method is static and can be invoked directly from a class.



Caution

common design error

It is a common design error to define an instance method that should have been defined static. For example, the method `factorial(int n)` should be defined static, as shown below, because it is independent of any specific instance.

```
public class Test {
  public int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

(a) Wrong design

```
public class Test {
  public static int factorial(int n)
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

(b) Correct design

8.8 Visibility Modifiers

You can use the `public` visibility modifier for classes, methods, and data fields to denote that they can be accessed from any other classes. If no visibility modifier is used, then by default the classes, methods, and data fields are accessible by any class in the same package. This is known as *package-private* or *package-access*.

using packages



Note

Packages can be used to organize classes. To do so, you need to add the following line as the first noncomment and nonblank statement in the program:

```
package packageName;
```

If a class is defined without the package statement, it is said to be placed in the *default package*.

Java recommends that you place classes into packages rather using a default package. For simplicity, however, this book uses default packages. For more information on packages, see Supplement III.G, "Packages."

In addition to the `public` and default visibility modifiers, Java provides the `private` and `protected` modifiers for class members. This section introduces the `private` modifier. The `protected` modifier will be introduced in §11.13, "The `protected` Data and Methods."

The `private` modifier makes methods and data fields accessible only from within its own class. Figure 8.13 illustrates how a public, default, and private data field or method in class `C1` can be accessed from a class `C2` in the same package and from a class `C3` in a different package.

```
package p1;

public class C1 {
  public int x;
  int y;
  private int z;

  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```
package p1;

public class C2 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    cannot access o.z;

    can invoke o.m1();
    can invoke o.m2();
    cannot invoke o.m3();
  }
}
```

```
package p2;

public class C3 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;

    can invoke o.m1();
    cannot invoke o.m2();
    cannot invoke o.m3();
  }
}
```

FIGURE 8.13 The private modifier restricts access to its defining class, the default modifier restricts access to a package, and the public modifier enables unrestricted access.

If a class is not defined public, it can be accessed only within the same package. As shown in Figure 8.14, `C1` can be accessed from `C2` but not from `C3`.

```
package p1;

class C1 {
  ...
}
```

```
package p1;

public class C2 {
  can access C1
}
```

```
package p2;

public class C3 {
  cannot access C1;
  can access C2;
}
```

FIGURE 8.14 A nonpublic class has package-access.

A visibility modifier specifies how data fields and methods in a class can be accessed from outside the class. There is no restriction on accessing data fields and methods from inside the

class. As shown in Figure 8.15(b), an object `foo` of the `Foo` class cannot access its private members, because `foo` is in the `Test` class. As shown in Figure 8.15(a), an object `foo` of the `Foo` class can access its private members, because `foo` is defined inside its own class.

inside access

```
public class Foo {
  private boolean x;

  public static void main(String[] args) {
    Foo foo = new Foo();
    System.out.println(foo.x);
    System.out.println(foo.convert());
  }

  private int convert() {
    return x ? 1 :  1;
  }
}
```

(a) This is OK because object `foo` is used inside the `Foo` class

```
public class Test {
  public static void main(String[] args) {
    Foo foo = new Foo();
    System.out.println(foo.x);
    System.out.println(foo.convert());
  }
}
```

(b) This is wrong because `x` and `convert` are private in `Foo`.

FIGURE 8.15 An object can access its private members if it is defined in its own class.



Caution

The `private` modifier applies only to the members of a class. The `public` modifier can apply to a class or members of a class. Using modifiers `public` and `private` on local variables would cause a compile error.



Note

In most cases, the constructor should be public. However, if you want to prohibit the user from creating an instance of a class, use a private constructor. For example, there is no reason to create an instance from the `Math` class, because all of its data fields and methods are static. To prevent the user from creating objects from the `Math` class, the constructor in `java.lang.Math` is defined as follows:

private constructor

```
private Math() {
}
```

8.9 Data Field Encapsulation



Video Note
Data field encapsulation

The data fields `radius` and `numberOfObjects` in the `Circle2` class in Listing 8.7 can be modified directly (e.g., `myCircle.radius = 5` or `Circle2.numberOfObjects = 10`). This is not a good practice—for two reasons:

- First, data may be tampered with. For example, `numberOfObjects` is to count the number of objects created, but it may be mistakenly set to an arbitrary value (e.g., `Circle2.numberOfObjects = 10`).
- Second, the class becomes difficult to maintain and vulnerable to bugs. Suppose you want to modify the `Circle2` class to ensure that the radius is nonnegative after other programs have already used the class. You have to change not only the `Circle2` class but also the programs that use it, because the clients may have modified the radius directly (e.g., `myCircle.radius = -5`).

To prevent direct modifications of data fields, you should declare the data fields private, using the `private` modifier. This is known as *data field encapsulation*.

data field encapsulation

A private data field cannot be accessed by an object from outside the class that defines the private field. But often a client needs to retrieve and modify a data field. To make a private

data field accessible, provide a *get* method to return its value. To enable a private data field to be updated, provide a *set* method to set a new value.



Note

Colloquially, a `get` method is referred to as a *getter* (or *accessor*), and a `set` method is referred to as a *setter* (or *mutator*).

accessor
mutator

A `get` method has the following signature:

```
public returnType getPropertyName()
```

boolean accessor

If the `returnType` is `boolean`, the `get` method should be defined as follows by convention:

```
public boolean isPropertyName()
```

A `set` method has the following signature:

```
public void setPropertyName(dataType propertyValue)
```

Let us create a new circle class with a private data-field radius and its associated accessor and mutator methods. The class diagram is shown in Figure 8.16. The new circle class, named `Circle3`, is defined in Listing 8.9:

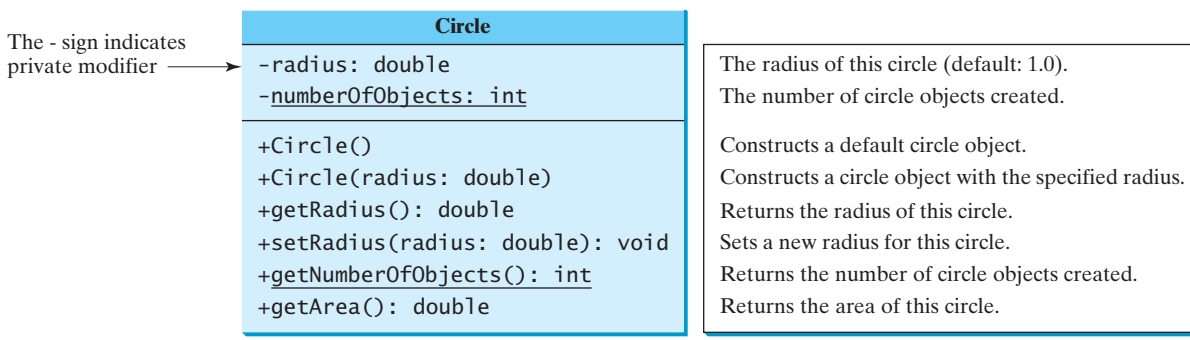

The - sign indicates private modifier
Circle
-radius: double
-numberOfObjects: int
+Circle()
+Circle(radius: double)
+getRadius(): double
+setRadius(radius: double): void
+getNumberOfObjects(): int
+getArea(): double
The radius of this circle (default: 1.0).
The number of circle objects created.
Constructs a default circle object.
Constructs a circle object with the specified radius.
Returns the radius of this circle.
Sets a new radius for this circle.
Returns the number of circle objects created.
Returns the area of this circle.

FIGURE 8.16 The `Circle` class encapsulates circle properties and provides get/set and other methods.

LISTING 8.9 Circle3.java

encapsulate **radius**

encapsulate **numberOfObjects**

```
1  public class Circle3 {
2    /** The radius of the circle */
3    private double radius = 1;
4
5    /** The number of the objects created */
6    private static int numberOfObjects = 0;
7
8    /** Construct a circle with radius 1 */
9    public Circle3() {
10     numberOfObjects++;
11   }
12
13   /** Construct a circle with a specified radius */
14   public Circle3(double newRadius) {
15     radius = newRadius;
16     numberOfObjects++;
17   }
18
```

```
19   /** Return radius */
20   public double getRadius() {                                    access method
21     return radius;
22   }
23
24   /** Set a new radius */
25   public void setRadius(double newRadius) {                      mutator method
26     radius = (newRadius >= 0) ? newRadius : 0;
27   }
28
29   /** Return numberOfObjects */
30   public static int getNumberOfObjects() {                       access method
31     return numberOfObjects;
32   }
33
34   /** Return the area of this circle */
35   public double getArea() {
36     return radius * radius * Math.PI;
37   }
38 }
```

The `getRadius()` method (lines 20–22) returns the radius, and the `setRadius(newRadius)` method (line 25–27) sets a new radius into the object. If the new radius is negative, `0` is set to the radius in the object. Since these methods are the only ways to read and modify radius, you have total control over how the `radius` property is accessed. If you have to change the implementation of these methods, you need not change the client programs. This makes the class easy to maintain.

Listing 8.10 gives a client program that uses the `Circle` class to create a `Circle` object and modifies the radius using the `setRadius` method.

LISTING 8.10 TestCircle3.java

```
1 public class TestCircle3 {
2   /** Main method */
3   public static void main(String[] args) {
4     // Create a Circle with radius 5.0
5     Circle3 myCircle = new Circle3(5.0);
6     System.out.println("The area of the circle of radius "
7       + myCircle.getRadius() + " is " + myCircle.getArea());      invoke public method
8
9     // Increase myCircle's radius by 10%
10    myCircle.setRadius(myCircle.getRadius() * 1.1);
11    System.out.println("The area of the circle of radius "
12      + myCircle.getRadius() + " is " + myCircle.getArea());      invoke public method
13
14    System.out.println("The number of objects created is "
15      + Circle3.getNumberOfObjects());                            invoke public method
16  }
17 }
```

The data field `radius` is declared private. Private data can be accessed only within their defining class. You cannot use `myCircle.radius` in the client program. A compile error would occur if you attempted to access private data from a client.

Since `numberOfObjects` is private, it cannot be modified. This prevents tampering. For example, the user cannot set `numberOfObjects` to `100`. The only way to make it `100` is to create `100` objects of the `Circle` class.

Suppose you combined `TestCircle` and `Circle` into one class by moving the `main` method in `TestCircle` into `Circle`. Could you use `myCircle.radius` in the `main` method? See Review Question 8.15 for the answer.



Design Guide

To prevent data from being tampered with and to make the class easy to maintain, declare data fields private.

8.10 Passing Objects to Methods

You can pass objects to methods. Like passing an array, passing an object is actually passing the reference of the object. The following code passes the `myCircle` object as an argument to the `printCircle` method:

```
1 public class Test {
2   public static void main(String[] args) {
3     // Circle3 is defined in Listing 8.9
4     Circle3 myCircle = new Circle3(5.0);
5     printCircle(myCircle);
6   }
7
8   public static void printCircle(Circle3 c) {
9     System.out.println("The area of the circle of radius "
10      + c.getRadius() + " is " + c.getArea());
11  }
12 }
```

pass an object

pass-by-value

Java uses exactly one mode of passing arguments: pass-by-value. In the preceding code, the value of `myCircle` is passed to the `printCircle` method. This value is a reference to a `Circle` object.

Let us demonstrate the difference between passing a primitive type value and passing a reference value with the program in Listing 8.11:

LISTING 8.11 TestPassObject.java

```
1 public class TestPassObject {
2   /** Main method */
3   public static void main(String[] args) {
4     // Create a Circle object with radius 1
5     Circle3 myCircle = new Circle3(1);
6
7     // Print areas for radius 1, 2, 3, 4, and 5.
8     int n = 5;
9     printAreas(myCircle, n);
10
11    // See myCircle.radius and times
12    System.out.println("\n" + "Radius is " + myCircle.getRadius());
13    System.out.println("n is " + n);
14  }
15
16  /** Print a table of areas for radius */
17  public static void printAreas(Circle3 c, int times) {
18    System.out.println("Radius \t\tArea");
19    while (times >= 1) {
20      System.out.println(c.getRadius() + "\t\t" + c.getArea());
21      c.setRadius(c.getRadius() + 1);
22      times--;
23    }
24  }
25 }
```

pass object

object parameter

```
Radius                          Area
 1.0                             3.141592653589793
 2.0                            12.566370614359172
 3.0                            29.274333882308138
 4.0                            50.26548245743669
 5.0                            79.53981633974483
Radius is 6.0
n is 5
```



The `Circle3` class is defined in Listing 8.9. The program passes a `Circle3` object `myCircle` and an integer value from `n` to invoke `printAreas(myCircle, n)` (line 9), which prints a table of areas for radii `1`, `2`, `3`, `4`, `5`, as shown in the sample output.

Figure 8.17 shows the call stack for executing the methods in the program. Note that the objects are stored in a heap.

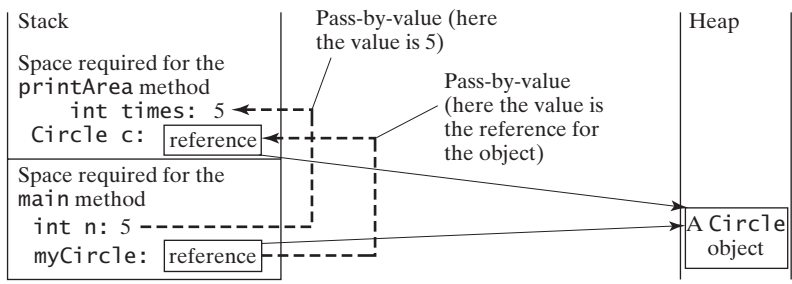


FIGURE 8.17 The value of `n` is passed to `times`, and the reference of `myCircle` is passed to `c` in the `printAreas` method.

When passing an argument of a primitive data type, the value of the argument is passed. In this case, the value of `n` (`5`) is passed to `times`. Inside the `printAreas` method, the content of `times` is changed; this does not affect the content of `n`.

When passing an argument of a reference type, the reference of the object is passed. In this case, `c` contains a reference for the object that is also referenced via `myCircle`. Therefore, changing the properties of the object through `c` inside the `printAreas` method has the same effect as doing so outside the method through the variable `myCircle`. Pass-by-value on references can be best described semantically as *pass-by-sharing*; i.e., the object referenced in the method is the same as the object being passed.

pass-by-sharing

8.11 Array of Objects

In Chapter 6, "Single-Dimensional Arrays," arrays of primitive type elements were created. You can also create arrays of objects. For example, the following statement declares and creates an array of ten `Circle` objects:

```
Circle[] circleArray = new Circle[10];
```

To initialize the `circleArray`, you can use a `for` loop like this one:

```
for (int i = 0; i < circleArray.length; i++) {
  circleArray[i] = new Circle();
}
```

An array of objects is actually an *array of reference variables*. So, invoking `circleArray[1].getArea()` involves two levels of referencing, as shown in Figure 8.18. `circleArray` references the entire array. `circleArray[1]` references a `Circle` object.

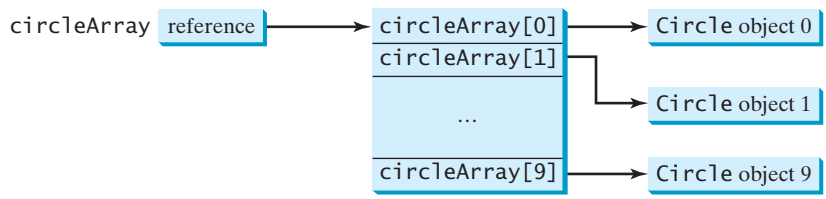


FIGURE 8.18 In an array of objects, an element of the array contains a reference to an object.



Note

When an array of objects is created using the `new` operator, each element in the array is a reference variable with a default value of `null`.

Listing 8.12 gives an example that demonstrates how to use an array of objects. The program summarizes the areas of an array of circles. The program creates `circleArray`, an array composed of five `Circle` objects; it then initializes circle radii with random values and displays the total area of the circles in the array.

LISTING 8.12 TotalArea.java

array of objects

return array of objects

pass array of objects

```
1  public class TotalArea {
2    /** Main method */
3    public static void main(String[] args) {
4      // Declare circleArray
5      Circle3[] circleArray;
6
7      // Create circleArray
8      circleArray = createCircleArray();
9
10     // Print circleArray and total areas of the circles
11     printCircleArray(circleArray);
12   }
13
14   /** Create an array of Circle objects */
15   public static Circle3[] createCircleArray() {
16     Circle3[] circleArray = new Circle3[5];
17
18     for (int i = 0; i < circleArray.length; i++) {
19       circleArray[i] = new Circle3(Math.random() * 100);
20     }
21
22     // Return Circle array
23     return circleArray;
24   }
25
26   /** Print an array of circles and their total area */
27   public static void printCircleArray(Circle3[] circleArray) {
28     System.out.printf("%-30s%-15s\n", "Radius", "Area");
29     for (int i = 0; i < circleArray.length; i++) {
30       System.out.printf("%-30f%-15f\n", circleArray[i].getRadius(),
31         circleArray[i].getArea());
32     }
33
34     System.out.println("----------------------------------------");
```

```
35
36      // Compute and display the result
37      System.out.printf("%-30s%-15f\n", "The total area of circles is",
38        sum(circleArray) );
39    }
40
41    /** Add circle areas */
42    public static double sum(Circle3[] circleArray) {
43      // Initialize sum
44      double sum = 0;
45
46      // Add areas to sum
47      for (int i = 0; i < circleArray.length; i++)
48        sum += circleArray[i].getArea();
49
50      return sum;
51    }
52  }
```

pass array of objects

```
 Radius                                    Area
70.577708                               15648.941866
44.152266                                6124.291736
24.867853                                1942.792644
 5.680718                                 101.380949
36.734246                                4239.280350
-------------------------------------------------------
The total area of circles is            28056.687544
```



The program invokes `createCircleArray()` (line 8) to create an array of five `Circle` objects. Several `Circle` classes were introduced in this chapter. This example uses the `Circle` class introduced in §8.9, "Data Field Encapsulation."

The circle radii are randomly generated using the `Math.random()` method (line 19). The `createCircleArray` method returns an array of `Circle` objects (line 23). The array is passed to the `printCircleArray` method, which displays the radius and area of each circle and the total area of the circles.

The sum of the circle areas is computed using the `sum` method (line 38), which takes the array of `Circle` objects as the argument and returns a `double` value for the total area.

Key Terms

accessor method (getter) 284
action 264
attribute 264
behavior 264
class 265
client 267
constructor 268
data field 268
data-field encapsulation 283
default constructor 270
dot operator (.) 271
instance 271
instance method 271
instance variable 271
instantiation 264
mutator method (setter) 285
`null` 272
no-arg constructor 266
object-oriented programming (OOP) 264
Unified Modeling Language (UML) 265
package-private (or package-access) 282
private 283
property 264
public 282
reference variable 271
reference type 271
state 264
static method 278
static variable 278

Chapter Summary

1. A class is a template for objects. It defines the properties of objects and provides constructors for creating objects and methods for manipulating them.
2. A class is also a data type. You can use it to declare object reference variables. An object reference variable that appears to hold an object actually contains a reference to that object. Strictly speaking, an object reference variable and an object are different, but most of the time the distinction can be ignored.
3. An object is an instance of a class. You use the `new` operator to create an object, and the dot (`.`) operator to access members of that object through its reference variable.
4. An instance variable or method belongs to an instance of a class. Its use is associated with individual instances. A static variable is a variable shared by all instances of the same class. A static method is a method that can be invoked without using instances.
5. Every instance of a class can access the class's static variables and methods. For clarity, however, it is better to invoke static variables and methods using `ClassName.variable` and `ClassName.method`.
6. Modifiers specify how the class, method, and data are accessed. A `public` class, method, or data is accessible to all clients. A `private` method or data is accessible only inside the class.
7. You can provide a `get` method or a `set` method to enable clients to see or modify the data. Colloquially, a `get` method is referred to as a *getter* (or *accessor*), and a `set` method as a *setter* (or *mutator*).
8. A `get` method has the signature `public returnType getPropertyName()`. If the `returnType` is `boolean`, the `get` method should be defined as `public boolean isPropertyName()`. A `set` method has the signature `public void setPropertyName(dataType propertyValue)`.
9. All parameters are passed to methods using pass-by-value. For a parameter of a primitive type, the actual value is passed; for a parameter of a reference type, the reference for the object is passed.
10. A Java array is an object that can contain primitive type values or object type values. When an array of objects is created, its elements are assigned the default value of `null`.

Review Questions

Sections 8.2–8.5

8.1 Describe the relationship between an object and its defining class. How do you define a class? How do you declare an object reference variable? How do you create an object? How do you declare and create an object in one statement?

8.2 What are the differences between constructors and methods?

8.3 Is an array an object or a primitive type value? Can an array contain elements of an object type as well as a primitive type? Describe the default value for the elements of an array.

8.4 What is wrong with the following program?

```
1 public class ShowErrors {
2   public static void main(String[] args) {
3     ShowErrors t = new ShowErrors(5);
4   }
5 }
```

(a)

```
1 public class ShowErrors {
2   public static void main(String[] args) {
3     ShowErrors t = new ShowErrors();
4     t.x();
5   }
6 }
```

(b)

```
1 public class ShowErrors {
2   public void method1() {
3     Circle c;
4     System.out.println("What is radius "
5       + c.getRadius());
6     c = new Circle();
7   }
8 }
```

(c)

```
1 public class ShowErrors {
2   public static void main(String[] args) {
3     C c = new C(5.0);
4     System.out.println(c.value);
5   }
6 }
7
8 class C {
9   int value = 2;
10 }
```

(d)

8.5 What is wrong in the following code?

```
1 class Test {
2   public static void main(String[] args) {
3     A a = new A();
4     a.print();
5   }
6 }
7
8 class A {
9   String s;
10
11   A(String s) {
12     this.s = s;
13   }
14
15   public void print() {
16     System.out.print(s);
17   }
18 }
```

8.6 What is the printout of the following code?

```
public class Foo {
  private boolean x;

  public static void main(String[] args) {
    Foo foo = new Foo();
    System.out.println(foo.x);
  }
}
```

Section 8.6

8.7 How do you create a `Date` for the current time? How do you display the current time?

8.8 How do you create a `JFrame`, set a title in a frame, and display a frame?

8.9 Which packages contain the classes `Date`, `JFrame`, `JOptionPane`, `System`, and `Math`?

Section 8.7

8.10 Suppose that the class `Foo` is defined in (a). Let `f` be an instance of `Foo`. Which of the statements in (b) are correct?

```
public class Foo {
  int i;
  static String s;

  void imethod() {
  }

  static void smethod() {
  }
}
```

(a)

```
System.out.println(f.i);
System.out.println(f.s);
f.imethod();
f.smethod();
System.out.println(Foo.i);
System.out.println(Foo.s);
Foo.imethod();
Foo.smethod();
```

(b)

8.11 Add the `static` keyword in the place of `?` if appropriate.

```
public class Test {
  private int count;

  public ? void main(String[] args) {
    ...
  }

  public ? int getCount() {
    return count;
  }

  public ? int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

8.12 Can you invoke an instance method or reference an instance variable from a static method? Can you invoke a static method or reference a static variable from an instance method? What is wrong in the following code?

```
 1 public class Foo {
 2   public static void main(String[] args) {
 3     method1();
 4   }
 5
 6   public void method1() {
 7     method2();
 8   }
 9
10   public static void method2() {
11     System.out.println("What is radius " + c.getRadius());
12   }
13
14   Circle c = new Circle();
15 }
```

Sections 8.8–8.9

8.13 What is an accessor method? What is a mutator method? What are the naming conventions for accessor methods and mutator methods?

8.14 What are the benefits of data-field encapsulation?

8.15 In the following code, `radius` is private in the `Circle` class, and `myCircle` is an object of the `Circle` class. Does the highlighted code below cause any problems? Explain why.

```
public class Circle {
  private double radius = 1.0;

  /** Find the area of this circle */
  public double getArea() {
    return radius * radius * Math.PI;
  }

  public static void main(String[] args) {
    Circle myCircle = new Circle();
    System.out.println("Radius is " + myCircle.radius);
  }
}
```

Section 8.10

8.16 Describe the difference between passing a parameter of a primitive type and passing a parameter of a reference type. Show the output of the following program:

```
public class Test {
  public static void main(String[] args) {
    Count myCount = new Count();
    int times = 0;

    for (int i = 0; i < 100; i++)
      increment(myCount, times);

    System.out.println("count is " + myCount.count);
    System.out.println("times is " + times);
  }

  public static void increment(Count c, int times) {
    c.count++;
    times++;
  }
}
```

```
public class Count {
  public int count;

  public Count(int c) {
    count = c;
  }

  public Count() {
    count = 1;
  }
}
```

8.17 Show the output of the following program:

```
public class Test {
  public static void main(String[] args) {
    Circle circle1 = new Circle(1);
    Circle circle2 = new Circle(2);

    swap1(circle1, circle2);
    System.out.println("After swap1: circle1 = " +
      circle1.radius + " circle2 = " + circle2.radius);

    swap2(circle1, circle2);
    System.out.println("After swap2: circle1 = " +
      circle1.radius + " circle2 = " + circle2.radius);
  }
```

```
  public static void swap1(Circle x, Circle y) {
    Circle temp = x;
    x = y;
    y = temp;
  }

  public static void swap2(Circle x, Circle y) {
    double temp = x.radius;
    x.radius = y.radius;
    y.radius = temp;
  }
}

class Circle {
  double radius;

  Circle(double newRadius) {
    radius = newRadius;
  }
}
```

8.18 Show the printout of the following code:

```
public class Test {
  public static void main(String[] args) {
    int[] a = {1, 2};
    swap(a[0], a[1]);
    System.out.println("a[0] = " + a[0]
      + " a[1] = " + a[1]);
  }

  public static void swap(int n1, int n2) {
    int temp = n1;
    n1 = n2;
    n2 = temp;
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    int[] a = {1, 2};
    swap(a);
    System.out.println("a[0] = " + a[0]
      + " a[1] = " + a[1]);
  }

  public static void swap(int[] a) {
    int temp = a[0];
    a[0] = a[1];
    a[1] = temp;
  }
}
```

(b)

```
public class Test {
  public static void main(String[] args) {
    T t = new T();
    swap(t);
    System.out.println("e1 = " + t.e1
      + " e2 = " + t.e2);
  }

  public static void swap(T t) {
    int temp = t.e1;
    t.e1 = t.e2;
    t.e2 = temp;
  }
}

class T {
  int e1 = 1;
  int e2 = 2;
}
```

(c)

```
public class Test {
  public static void main(String[] args) {
    T t1 = new T();
    T t2 = new T();
    System.out.println("t1's i = " +
      t1.i + " and j = " + t1.j);
    System.out.println("t2's i = " +
      t2.i + " and j = " + t2.j);
  }
}

class T {
  static int i = 0;
  int j = 0;

  T() {
    i++;
    j = 1;
  }
}
```

(d)

8.19 What is the output of the following program?

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = null;
    m1(date);
    System.out.println(date);
  }

  public static void m1(Date date) {
    date = new Date();
  }
}
```

(a)

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date = new Date(7654321);
  }
}
```

(b)

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date.setTime(7654321);
  }
}
```

(c)

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date = null;
  }
}
```

(d)

Section 8.11

8.20 What is wrong in the following code?

```
1 public class Test {
2   public static void main(String[] args) {
3     java.util.Date[] dates = new java.util.Date[10];
4     System.out.println(dates[0]);
5     System.out.println(dates[0].toString());
6   }
7 }
```

Programming Exercises



Pedagogical Note

The exercises in Chapters 8–14 achieve three objectives:

three objectives

- Design classes and draw UML class diagrams;
- Implement classes from the UML;
- Use classes to develop applications.

Solutions for the UML diagrams for the even-numbered exercises can be downloaded from the Student Website and all others can be downloaded from the Instructor Website.

Sections 8.2–8.5

8.1 (*The `Rectangle` class*) Following the example of the `Circle` class in §8.2, design a class named `Rectangle` to represent a rectangle. The class contains:

- Two `double` data fields named `width` and `height` that specify the width and height of the rectangle. The default values are `1` for both `width` and `height`.
- A no-arg constructor that creates a default rectangle.
- A constructor that creates a rectangle with the specified `width` and `height`.
- A method named `getArea()` that returns the area of this rectangle.
- A method named `getPerimeter()` that returns the perimeter.

Draw the UML diagram for the class. Implement the class. Write a test program that creates two `Rectangle` objects—one with width `4` and height `40` and the other with width `3.5` and height `35.9`. Display the width, height, area, and perimeter of each rectangle in this order.

8.2 (*The `Stock` class*) Following the example of the `Circle` class in §8.2, design a class named `Stock` that contains:

- A string data field named `symbol` for the stock's symbol.
- A string data field named `name` for the stock's name.
- A `double` data field named `previousClosingPrice` that stores the stock price for the previous day.
- A `double` data field named `currentPrice` that stores the stock price for the current time.
- A constructor that creates a stock with specified symbol and name.
- A method named `getChangePercent()` that returns the percentage changed from `previousClosingPrice` to `currentPrice`.

Draw the UML diagram for the class. Implement the class. Write a test program that creates a `Stock` object with the stock symbol JAVA, the name Sun Microsystems Inc, and the previous closing price of `4.5`. Set a new current price to `4.35` and display the price-change percentage.

Section 8.6

8.3* (*Using the `Date` class*) Write a program that creates a `Date` object, sets its elapsed time to `10000`, `100000`, `10000000`, `10000000`, `100000000`, `1000000000`, `10000000000`, `100000000000`, and displays the date and time using the `toString()` method, respectively.

8.4* (*Using the `Random` class*) Write a program that creates a `Random` object with seed `1000` and displays the first 50 random integers between `0` and `100` using the `nextInt(100)` method.

8.5* (*Using the `GregorianCalendar` class*) Java API has the `GregorianCalendar` class in the `java.util` package that can be used to obtain the year, month, and day of a date. The no-arg constructor constructs an instance for the current date, and the methods `get(GregorianCalendar.YEAR)`, `get(GregorianCalendar.MONTH)`, and `get(GregorianCalendar.DAY_OF_MONTH)` return the year, month, and day. Write a program to perform two tasks:

- Display the current year, month, and day.
- The `GregorianCalendar` class has the `setTimeInMillis(long)`, which can be used to set a specified elapsed time since January 1, 1970. Set the value to `1234567898765L` and display the year, month, and day.

Sections 8.7–8.9

8.6** (*Displaying calendars*) Rewrite the `PrintCalendar` class in Listing 5.12 to display calendars in a message dialog box. Since the output is generated from several static methods in the class, you may define a static `String` variable `output` for storing the output and display it in a message dialog box.

8.7 (*The `Account` class*) Design a class named `Account` that contains:

- A private `int` data field named `id` for the account (default `0`).
- A private `double` data field named `balance` for the account (default `0`).
- A private `double` data field named `annualInterestRate` that stores the current interest rate (default `0`). Assume all accounts have the same interest rate.
- A private `Date` data field named `dateCreated` that stores the date when the account was created.
- A no-arg constructor that creates a default account.
- A constructor that creates an account with the specified id and initial balance.
- The accessor and mutator methods for `id`, `balance`, and `annualInterestRate`.
- The accessor method for `dateCreated`.
- A method named `getMonthlyInterestRate()` that returns the monthly interest rate.
- A method named `withdraw` that withdraws a specified amount from the account.
- A method named `deposit` that deposits a specified amount to the account.

Draw the UML diagram for the class. Implement the class. Write a test program that creates an `Account` object with an account ID of 1122, a balance of $20,000, and an annual interest rate of 4.5%. Use the `withdraw` method to withdraw $2,500, use the `deposit` method to deposit $3,000, and print the balance, the monthly interest, and the date when this account was created.

8.8 (*The `Fan` class*) Design a class named `Fan` to represent a fan. The class contains:



Video Note
The Fan class

- Three constants named `SLOW`, `MEDIUM`, and `FAST` with values `1`, `2`, and `3` to denote the fan speed.
- A private `int` data field named `speed` that specifies the speed of the fan (default `SLOW`).
- A private `boolean` data field named `on` that specifies whether the fan is on (default `false`).
- A private `double` data field named `radius` that specifies the radius of the fan (default `5`).
- A string data field named `color` that specifies the color of the fan (default `blue`).
- The accessor and mutator methods for all four data fields.
- A no-arg constructor that creates a default fan.
- A method named `toString()` that returns a string description for the fan. If the fan is on, the method returns the fan speed, color, and radius in one combined string. If the fan is not on, the method returns fan color and radius along with the string "fan is off" in one combined string.

Draw the UML diagram for the class. Implement the class. Write a test program that creates two `Fan` objects. Assign maximum speed, radius `10`, color `yellow`, and turn it on to the first object. Assign medium speed, radius `5`, color `blue`, and turn it off to the second object. Display the objects by invoking their `toString` method.

8.9** (*Geometry: n-sided regular polygon*) In an *n*-sided regular polygon all sides have the same length and all angles have the same degree (i.e., the polygon is

both equilateral and equiangular). Design a class named `RegularPolygon` that contains:

- A private `int` data field named `n` that defines the number of sides in the polygon with default value `3`.
- A private `double` data field named `side` that stores the length of the side with default value `1`.
- A private `double` data field named `x` that defines the *x*-coordinate of the center of the polygon with default value `0`.
- A private `double` data field named `y` that defines the *y*-coordinate of the center of the polygon with default value `0`.
- A no-arg constructor that creates a regular polygon with default values.
- A constructor that creates a regular polygon with the specified number of sides and length of side, centered at (`0`, `0`).
- A constructor that creates a regular polygon with the specified number of sides, length of side, and *x*-and *y*-coordinates.
- The accessor and mutator methods for all data fields.
- The method `getPerimeter()` that returns the perimeter of the polygon.
- The method `getArea()` that returns the area of the polygon. The formula for computing the area of a regular polygon is

$$Area = \frac{n \times s^2}{4 \times \tan\left(\frac{\pi}{n}\right)}.$$

Draw the UML diagram for the class. Implement the class. Write a test program that creates three `RegularPolygon` objects, created using the no-arg constructor, using `RegularPolygon(6, 4)`, and using `RegularPolygon(10, 4, 5.6, 7.8)`. For each object, display its perimeter and area.

8.10* (*Algebra: quadratic equations*) Design a class named `QuadraticEquation` for a quadratic equation $ax^2 + bx + x = 0$. The class contains:

- Private data fields `a`, `b`, and `c` that represents three coefficients.
- A constructor for the arguments for `a`, `b`, and `c`.
- Three `get` methods for `a`, `b`, and `c`.
- A method named `getDiscriminant()` that returns the discriminant, which is $b^2 - 4ac$.
- The methods named `getRoot1()` and `getRoot2()` for returning two roots of the equation

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \text{ and } r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

These methods are useful only if the discriminant is nonnegative. Let these methods return `0` if the discriminant is negative.

Draw the UML diagram for the class. Implement the class. Write a test program that prompts the user to enter values for *a*, *b*, and *c* and displays the result based on the discriminant. If the discriminant is positive, display the two roots. If the discriminant is `0`, display the one root. Otherwise, display "The equation has no roots." See Exercise 3.1 for sample runs.

8.11* (*Algebra: 2 × 2 linear equations*) Design a class named **LinearEquation** for a 2 × 2 system of linear equations:

$$ax + by = e \qquad cx + dy = f$$

$$x = \frac{ed - bf}{ad - bc} \qquad y = \frac{af - ec}{ad - bc}$$

The class contains:

- Private data fields **a**, **b**, **c**, **d**, **e**, and **f**.
- A constructor with the arguments for **a**, **b**, **c**, **d**, **e**, and **f**.
- Six **get** methods for **a**, **b**, **c**, **d**, **e**, and **f**.
- A method named **isSolvable()** that returns true if $ad - bc$ is not **0**.
- Methods **getX()** and **getY()** that return the solution for the equation.

Draw the UML diagram for the class. Implement the class. Write a test program that prompts the user to enter **a**, **b**, **c**, **d**, **e**, and **f** and displays the result. If $ad - bc$ is **0**, report that "The equation has no solution." See Exercise 3.3 for sample runs.

8.12** (*Geometry: intersection*) Suppose two line segments intersect. The two endpoints for the first line segment are (**x1**, **y1**) and (**x2**, **y2**) and for the second line segment are (**x3**, **y3**) and (**x4**, **y5**). Write a program that prompts the user to enter these four endpoints and displays the intersecting point.

(*Hint*: Use the **LinearEquation** class from the preceding exercise.)

```
Enter the endpoints of the first line segment: 2.0 2.0 0 0 Enter
Enter the endpoints of the second line segment: 0 2.0 2.0 0 Enter
The intersecting point is: (1.0, 1.0)
```



8.13** (*The **Location** class*) Design a class named **Location** for locating a maximal value and its location in a two-dimensional array. The class contains public data fields **row**, **column**, and **maxValue** that store the maximal value and its indices in a two dimensional array with **row** and **column** as **int** type and **maxValue** as **double** type.

Write the following method that returns the location of the largest element in a two-dimensional array.

```
public static Location locateLargest(double[][] a)
```

The return value is an instance of **Location**. Write a test program that prompts the user to enter a two-dimensional array and displays the location of the largest element in the array. Here is a sample run:

```
Enter the number of rows and columns of the array: 3 4 Enter
Enter the array:
23.5 35 2 10 Enter
4.5 3 45 3.5 Enter
35 44 5.5 9.6 Enter
The location of the largest element is 45 at (1, 2)
```



This page intentionally left blank

CHAPTER 9

STRINGS AND TEXT I/O

Objectives

- To use the `String` class to process fixed strings (§9.2).
- To use the `Character` class to process a single character (§9.3).
- To use the `StringBuilder`/`StringBuffer` class to process flexible strings (§9.4).
- To distinguish among the `String`, `StringBuilder`, and `StringBuffer` classes (§9.2–9.4).
- To learn how to pass arguments to the `main` method from the command line (§9.5).
- To discover file properties and to delete and rename files using the `File` class (§9.6).
- To write data to a file using the `PrintWriter` class (§9.7.1).
- To read data from a file using the `Scanner` class (§9.7.2).
- (GUI) To open files using a dialog box (§9.8).



9.1 Introduction

problem

Often you encounter problems that involve string processing and file input and output. Suppose you need to write a program that replaces all occurrences of a word in a file with a new word. How do you accomplish this? This chapter introduces strings and text files, which will enable you to solve problems of this type. (Since no new concepts are introduced here, instructors may assign this chapter for students to study on their own.)

9.2 The `String` Class

A *string* is a sequence of characters. In many languages, strings are treated as an array of characters, but in Java a string is an object. The `String` class has 11 constructors and more than 40 methods for manipulating strings. Not only is it very useful in programming, but also it is a good example for learning classes and objects.

9.2.1 Constructing a String

You can create a string object from a string literal or from an array of characters. To create a string from a string literal, use a syntax like this one:

```
String newString = new String(stringLiteral);
```

The argument `stringLiteral` is a sequence of characters enclosed inside double quotes. The following statement creates a `String` object `message` for the string literal `"Welcome to Java"`:

```
String message = new String("Welcome to Java");
```

string literal object

Java treats a string literal as a `String` object. So, the following statement is valid:

```
String message = "Welcome to Java";
```

You can also create a string from an array of characters. For example, the following statements create the string "Good Day":

```
char[] charArray = {'G', 'o', 'o', 'd', ' ', 'D', 'a', 'y'};
String message = new String(charArray);
```



Note

A `String` variable holds a reference to a `String` object that stores a string value. Strictly speaking, the terms *`String` variable*, *`String` object*, and *string value* are different, but most of the time the distinctions between them can be ignored. For simplicity, the term *string* will often be used to refer to `String` variable, `String` object, and string value.

string variable, string object, string value

9.2.2 Immutable Strings and Interned Strings

immutable

A `String` object is immutable; its contents cannot be changed. Does the following code change the contents of the string?

```
String s = "Java";
s = "HTML";
```

The answer is no. The first statement creates a `String` object with the content "Java" and assigns its reference to `s`. The second statement creates a new `String` object with the content "HTML" and assigns its reference to `s`. The first `String` object still exists after the assignment, but it can no longer be accessed, because variable `s` now points to the new object, as shown in Figure 9.1.

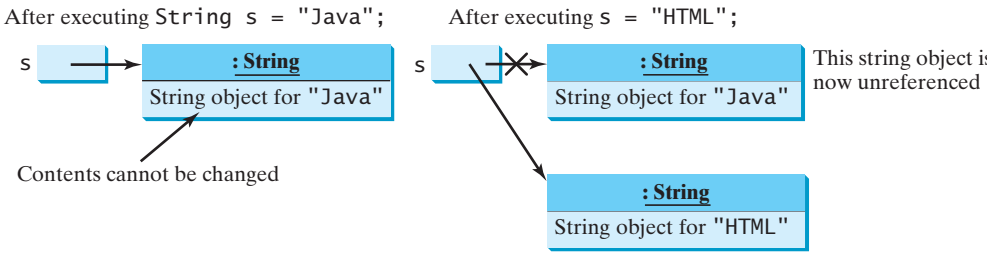
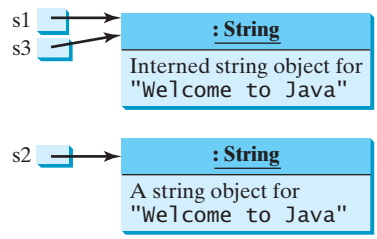


FIGURE 9.1 Strings are immutable; once created, their contents cannot be changed.

Since strings are immutable and are ubiquitous in programming, the JVM uses a unique instance for string literals with the same character sequence in order to improve efficiency and save memory. Such an instance is called *interned*. For example, the following statements: interned string

```
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";
System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```



display

```
s1 == s2 is false
s1 == s3 is true
```

In the preceding statements, `s1` and `s3` refer to the same interned string "Welcome to Java", therefore `s1 == s3` is `true`. However, `s1 == s2` is `false`, because `s1` and `s2` are two different string objects, even though they have the same contents.

9.2.3 String Comparisons

The `String` class provides the methods for comparing strings, as shown in Figure 9.2.

java.lang.String	
+equals(s1: String): boolean	Returns true if this string is equal to string s1.
+equalsIgnoreCase(s1: String): boolean	Returns true if this string is equal to string s1 case insensitive.
+compareTo(s1: String): int	Returns an integer greater than 0, equal to 0, or less than 0 to indicate whether this string is greater than, equal to, or less than s1.
+compareToIgnoreCase(s1: String): int	Same as compareTo except that the comparison is case insensitive.
+regionMatches(index: int, s1: String, s1Index: int, len: int): boolean	Returns true if the specified subregion of this string exactly matches the specified subregion in string s1.
+regionMatches(ignoreCase: boolean, index: int, s1: String, s1Index: int, len: int): boolean	Same as the preceding method except that you can specify whether the match is case sensitive.
+startsWith(prefix: String): boolean	Returns true if this string starts with the specified prefix.
+endsWith(suffix: String): boolean	Returns true if this string ends with the specified suffix.

FIGURE 9.2 The `String` class contains the methods for comparing strings.

How do you compare the contents of two strings? You might attempt to use the `==` operator, as follows:

`==`

```
if (string1 == string2)
  System.out.println("string1 and string2 are the same object");
else
  System.out.println("string1 and string2 are different objects");
```

However, the `==` operator checks only whether `string1` and `string2` refer to the same object; it does not tell you whether they have the same contents. Therefore, you cannot use the `==` operator to find out whether two string variables have the same contents. Instead, you should use the `equals` method. The code given below, for instance, can be used to compare two strings:

`string1.equals(string2)`

```
if (string1.equals(string2))
  System.out.println("string1 and string2 have the same contents");
else
  System.out.println("string1 and string2 are not equal");
```

For example, the following statements display `true` and then `false`.

```
String s1 = new String("Welcome to Java");
String s2 = "Welcome to Java";
String s3 = "Welcome to C++";
System.out.println(s1.equals(s2)); // true
System.out.println(s1.equals(s3)); // false
```

The `compareTo` method can also be used to compare two strings. For example, consider the following code:

`s1.compareTo(s2)`

```
s1.compareTo(s2)
```

The method returns the value `0` if `s1` is equal to `s2`, a value less than `0` if `s1` is lexicographically (i.e., in terms of Unicode ordering) less than `s2`, and a value greater than `0` if `s1` is lexicographically greater than `s2`.

The actual value returned from the `compareTo` method depends on the offset of the first two distinct characters in `s1` and `s2` from left to right. For example, suppose `s1` is `"abc"` and `s2` is `"abg"`, and `s1.compareTo(s2)` returns `-4`. The first two characters (`a` vs. `a`) from `s1` and `s2` are compared. Because they are equal, the second two characters (`b` vs. `b`) are compared. Because they are also equal, the third two characters (`c` vs. `g`) are compared. Since the character `c` is `4` less than `g`, the comparison returns `-4`.



Caution

Syntax errors will occur if you compare strings by using comparison operators, such as `>`, `>=`, `<`, or `<=`. Instead, you have to use `s1.compareTo(s2)`.



Note

The `equals` method returns `true` if two strings are equal and `false` if they are not. The `compareTo` method returns `0`, a positive integer, or a negative integer, depending on whether one string is equal to, greater than, or less than the other string.

The `String` class also provides `equalsIgnoreCase`, `compareToIgnoreCase`, and `regionMatches` methods for comparing strings. The `equalsIgnoreCase` and `compareToIgnoreCase` methods ignore the case of the letters when comparing two strings. The `regionMatches` method compares portions of two strings for equality. You can also use `str.startsWith(prefix)` to check whether string `str` starts with a specified prefix, and `str.endsWith(suffix)` to check whether string `str` ends with a specified `suffix`.

9.2.4 String Length, Characters, and Combining Strings

The `String` class provides the methods for obtaining length, retrieving individual characters, and concatenating strings, as shown in Figure 9.3.

java.lang.String	
`+length(): int`	Returns the number of characters in this string.
`+charAt(index: int): char`	Returns the character at the specified index from this string.
`+concat(s1: String): String`	Returns a new string that concatenates this string with string `s1`.

FIGURE 9.3 The `String` class contains the methods for getting string length, individual characters, and combining strings.

You can get the length of a string by invoking its `length()` method. For example, `message.length()` returns the length of the string `message`.

`length()`



Caution

`length` is a method in the `String` class but is a property of an array object. So you have to use `s.length()` to get the number of characters in string `s`, and `a.length` to get the number of elements in array `a`.

`length()`

The `s.charAt(index)` method can be used to retrieve a specific character in a string `s`, where the index is between `0` and `s.length()-1`. For example, `message.charAt(0)` returns the character `W`, as shown in Figure 9.4.

`charAt(index)`

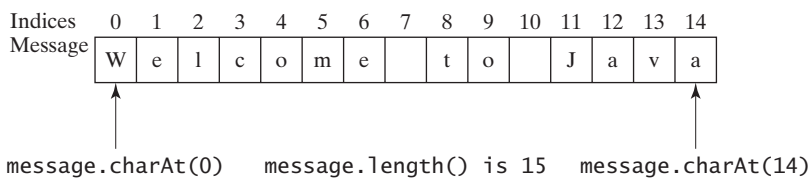


FIGURE 9.4 A `String` object is represented using an array internally.



Note

When you use a string, you often know its literal value. For convenience, Java allows you to use the string literal to refer directly to strings without creating new variables. Thus, `"Welcome to Java".charAt(0)` is correct and returns `W`.

string literal



Note

A string value is represented using a private array variable internally. The array cannot be accessed outside of the `String` class. The `String` class provides many public methods, such as `length()` and `charAt(index)`, to retrieve the array information. This is a good example of encapsulation: the data field of the class is hidden from the user through the private modifier, and thus the user cannot directly manipulate it. If the array were not private, the user would be able to change the string content by modifying the array. This would violate the tenet that the `String` class is immutable.

encapsulating string



Caution

Attempting to access characters in a string `s` out of bounds is a common programming error. To avoid it, make sure that you do not use an index beyond `s.length() - 1`. For example, `s.charAt(s.length())` would cause a `StringIndexOutOfBoundsException`.

string index range

You can use the `concat` method to concatenate two strings. The statement shown below, for example, concatenates strings `s1` and `s2` into `s3`:

`s1.concat(s2)`

```
String s3 = s1.concat(s2);
```

Since string concatenation is heavily used in programming, Java provides a convenient way to accomplish it. You can use the plus (+) sign to concatenate two or more strings. So the above statement is equivalent to

`s1 + s2`

```
String s3 = s1 + s2;
```

The following code combines the strings `message`, `" and "`, and `"HTML"` into one string:

```
String myString = message + " and " + "HTML";
```

Recall that the + sign can also concatenate a number with a string. In this case, the number is converted into a string and then concatenated. Note that at least one of the operands must be a string in order for concatenation to take place.

9.2.5 Obtaining Substrings

You can obtain a single character from a string using the `charAt` method, as shown in Figure 9.3. You can also obtain a substring from a string using the `substring` method in the `String` class, as shown in Figure 9.5.

java.lang.String	
`+substring(beginIndex: int): String`	Returns this string's substring that begins with the character at the specified `beginIndex` and extends to the end of the string, as shown in Figure 9.6.
`+substring(beginIndex: int, endIndex: int): String`	Returns this string's substring that begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`, as shown in Figure 9.6. Note that the character at `endIndex` is not part of the substring.

FIGURE 9.5 The `String` class contains the methods for obtaining substrings.

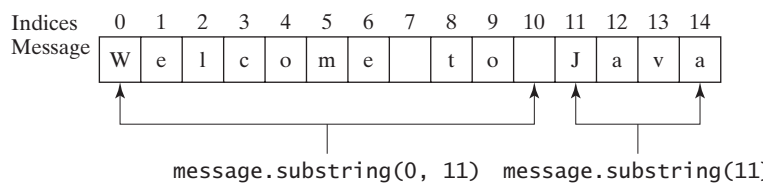


FIGURE 9.6 The `substring` method obtains a substring from a string.

For example,

```
String message = "Welcome to Java".substring(0, 11) + "HTML";
```

The string `message` now becomes `"Welcome to HTML"`.

`beginIndex <= endIndex`

Note

If `beginIndex` is `endIndex`, `substring(beginIndex, endIndex)` returns an empty string with length `0`. If `beginIndex` > `endIndex`, it would be a runtime error.

9.2.6 Converting, Replacing, and Splitting Strings

The `String` class provides the methods for converting, replacing, and splitting strings, as shown in Figure 9.7.

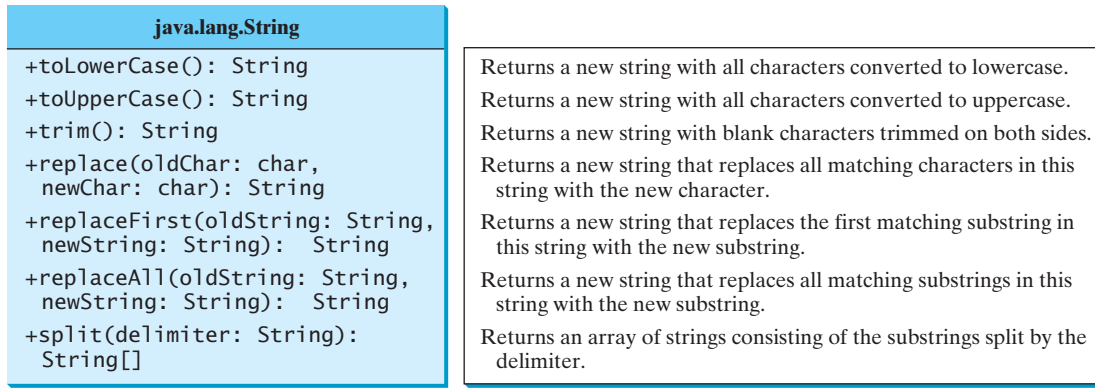

java.lang.String
+toLowerCase(): String
+toUpperCase(): String
+trim(): String
+replace(oldChar: char, newChar: char): String
+replaceFirst(oldString: String, newString: String): String
+replaceAll(oldString: String, newString: String): String
+split(delimiter: String): String[]
Returns a new string with all characters converted to lowercase.
Returns a new string with all characters converted to uppercase.
Returns a new string with blank characters trimmed on both sides.
Returns a new string that replaces all matching characters in this string with the new character.
Returns a new string that replaces the first matching substring in this string with the new substring.
Returns a new string that replaces all matching substrings in this string with the new substring.
Returns an array of strings consisting of the substrings split by the delimiter.

FIGURE 9.7 The `String` class contains the methods for converting, replacing, and splitting strings.

Once a string is created, its contents cannot be changed. The methods `toLowerCase`, `toUpperCase`, `trim`, `replace`, `replaceFirst`, and `replaceAll` return a new string derived from the original string (without changing the original string!). The `toLowerCase` and `toUpperCase` methods return a new string by converting all the characters in the string to lowercase or uppercase. The `trim` method returns a new string by eliminating blank characters from both ends of the string. Several versions of the `replace` methods are provided to replace a character or a substring in the string with a new character or a new substring.

For example,

```
"Welcome".toLowerCase() returns a new string, welcome.                    toLowerCase()
"Welcome".toUpperCase() returns a new string, WELCOME.                    toUpperCase()
"  Welcome  ".trim() returns a new string, Welcome.                       trim()
"Welcome".replace('e', 'A') returns a new string, WAlcomA.                replace
"Welcome".replaceFirst("e", "AB") returns a new string, WABlcome.         replaceFirst
"Welcome".replace("e", "AB") returns a new string, WABlcomAB.             replace
"Welcome".replace("el", "AB") returns a new string, WABcome.              replace
```

The `split` method can be used to extract tokens from a string with the specified delimiters. For example, the following code

`split`

```
String[] tokens = "Java#HTML#Perl".split("#", 0);
for (int i = 0; i < tokens.length; i++)
  System.out.print(tokens[i] + " ");
```

displays

Java HTML Perl

9.2.7 Matching, Replacing and Splitting by Patterns

You can match, replace, or split a string by specifying a pattern. This is an extremely useful and powerful feature, commonly known as *regular expression*. Regular expressions seem complex to beginning students. For this reason, two simple patterns are used in this section. Please refer to Supplement III.H, "Regular Expressions," for further studies.

regular expression

matches(regex)

Let us begin with the `matches` method in the `String` class. At first glance, the `matches` method is very similar to the `equals` method. For example, the following two statements both evaluate to `true`.

```
"Java".matches("Java");
"Java".equals("Java");
```

However, the `matches` method is more powerful. It can match not only a fixed string, but also a set of strings that follow a pattern. For example, the following statements all evaluate to `true`:

```
"Java is fun".matches("Java.*")
"Java is cool".matches("Java.*")
"Java is powerful".matches("Java.*")
```

`"Java.*"` in the preceding statements is a regular expression. It describes a string pattern that begins with Java followed by *any* zero or more characters. Here, the substring `.*` matches any zero or more characters.

The `replaceAll`, `replaceFirst`, and `split` methods can be used with a regular expression. For example, the following statement returns a new string that replaces `$`, `+`, or `#` in `"a+b$#c"` with the string `NNN`.

replaceAll(regex)

```
String s = "a+b$#c".replaceAll("[$+#]", "NNN");
System.out.println(s);
```

Here the regular expression `[$+#]` specifies a pattern that matches `$`, `+`, or `#`. So, the output is `aNNNbNNNNNNc`.

The following statement splits the string into an array of strings delimited by punctuation marks.

split(regex)

```
String[] tokens = "Java,C?C#,C++".split("[.,:;?]");

for (int i = 0; i < tokens.length; i++)
  System.out.println(tokens[i]);
```

Here the regular expression `[.,:;?]` specifies a pattern that matches `.`, `,`, `:`, `;`, or `?`. Each of these characters is a delimiter for splitting the string. So, the string is split into `Java`, `C`, `C#`, and `C++`, which are stored into array `tokens`.

9.2.8 Finding a Character or a Substring in a String

The `String` class provides several overloaded `indexOf` and `lastIndexOf` methods to find a character or a substring in a string, as shown in Figure 9.8.

For example,

indexOf

```
"Welcome to Java".indexOf('W') returns 0.
"Welcome to Java".indexOf('o') returns 4.
"Welcome to Java".indexOf('o', 5) returns 9.
"Welcome to Java".indexOf("come") returns 3.
"Welcome to Java".indexOf("Java", 5) returns 11.
"Welcome to Java".indexOf("java", 5) returns -1.
```

lastIndexOf

```
"Welcome to Java".lastIndexOf('W') returns 0.
"Welcome to Java".lastIndexOf('o') returns 9.
"Welcome to Java".lastIndexOf('o', 5) returns 4.
"Welcome to Java".lastIndexOf("come") returns 3.
"Welcome to Java".lastIndexOf("Java", 5) returns -1.
"Welcome to Java".lastIndexOf("Java") returns 11.
```

java.lang.String	
+indexOf(ch: char): int	Returns the index of the first occurrence of ch in the string. Returns -1 if not matched.
+indexOf(ch: char, fromIndex: int): int	Returns the index of the first occurrence of ch after fromIndex in the string. Returns -1 if not matched.
+indexOf(s: String): int	Returns the index of the first occurrence of string s in this string. Returns -1 if not matched.
+indexOf(s: String, fromIndex: int): int	Returns the index of the first occurrence of string s in this string after fromIndex. Returns -1 if not matched.
+lastIndexOf(ch: int): int	Returns the index of the last occurrence of ch in the string. Returns -1 if not matched.
+lastIndexOf(ch: int, fromIndex: int): int	Returns the index of the last occurrence of ch before fromIndex in this string. Returns -1 if not matched.
+lastIndexOf(s: String): int	Returns the index of the last occurrence of string s. Returns -1 if not matched.
+lastIndexOf(s: String, fromIndex: int): int	Returns the index of the last occurrence of string s before fromIndex. Returns -1 if not matched.

FIGURE 9.8 The `String` class contains the methods for matching substrings.

9.2.9 Conversion between Strings and Arrays

Strings are not arrays, but a string can be converted into an array, and vice versa. To convert a string to an array of characters, use the `toCharArray` method. For example, the following statement converts the string `"Java"` to an array.

`toCharArray`

```
char[] chars = "Java".toCharArray();
```

So `chars[0]` is `'J'`, `chars[1]` is `'a'`, `chars[2]` is `'v'`, and `chars[3]` is `'a'`.

You can also use the `getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)` method to copy a substring of the string from index `srcBegin` to index `srcEnd-1` into a character array `dst` starting from index `dstBegin`. For example, the following code copies a substring "3720" in "CS3720" from index `2` to index `6-1` into the character array `dst` starting from index `4`.

`getChars`

```
char[]  dst = {'J', 'A', 'V', 'A', '1', '3', '0', '1'};
"CS3720".getChars(2, 6, dst, 4);
```

Thus `dst` becomes {`'J'`, `'A'`, `'V'`, `'A'`, `'3'`, `'7'`, `'2'`, `'0'`}.

To convert an array of characters into a string, use the `String(char[])` constructor or the `valueOf(char[])` method. For example, the following statement constructs a string from an array using the `String` constructor.

```
String str = new String(new char[]{'J', 'a', 'v', 'a'});
```

The next statement constructs a string from an array using the `valueOf` method.

`valueOf`

```
String str = String.valueOf(new char[]{'J', 'a', 'v', 'a'});
```

9.2.10 Converting Characters and Numeric Values to Strings

The static `valueOf` method can be used to convert an array of characters into a string. There are several overloaded versions of the `valueOf` method that can be used to convert a character and numeric values to strings with different parameter types, `char`, `double`, `long`, `int`, and `float`, as shown in Figure 9.9.

overloaded `valueOf`

java.lang.String	
+valueOf(c: char): String	Returns a string consisting of the character c.
+valueOf(data: char[]): String	Returns a string consisting of the characters in the array.
+valueOf(d: double): String	Returns a string representing the double value.
+valueOf(f: float): String	Returns a string representing the float value.
+valueOf(i: int): String	Returns a string representing the int value.
+valueOf(l: long): String	Returns a string representing the long value.
+valueOf(b: boolean): String	Returns a string representing the boolean value.

FIGURE 9.9 The `String` class contains the static methods for creating strings from primitive type values.

For example, to convert a `double` value `5.44` to a string, use `String.valueOf(5.44)`. The return value is a string consisting of the characters `'5'`, `'.'`, `'4'`, and `'4'`.



Note

Use `Double.parseDouble(str)` or `Integer.parseInt(str)` to convert a string to a `double` value or an `int` value.

9.2.11 Formatting Strings

The `String` class contains the static `format` method in the `String` class to create a formatted string. The syntax to invoke this method is

```
String.format(format, item1, item2, ..., itemk)
```

This method is similar to the `printf` method except that the `format` method returns a formatted string, whereas the `printf` method displays a formatted string. For example,

```
String s = String.format("%5.2f", 45.556);
```

creates a formatted string `"45.56"`.

9.2.12 Problem: Checking Palindromes



Video Note
Check palindrome

A string is a palindrome if it reads the same forward and backward. The words "mom," "dad," and "noon," for instance, are all palindromes.

The problem is to write a program that prompts the user to enter a string and reports whether the string is a palindrome. One solution is to check whether the first character in the string is the same as the last character. If so, check whether the second character is the same as the second-to-last character. This process continues until a mismatch is found or all the characters in the string are checked, except for the middle character if the string has an odd number of characters.

To implement this idea, use two variables, say `low` and `high`, to denote the position of two characters at the beginning and the end in a string `s`, as shown in Listing 9.1 (lines 22, 25). Initially, `low` is `0` and `high` is `s.length() - 1`. If the two characters at these positions match, increment `low` by `1` and decrement `high` by `1` (lines 31–32). This process continues until (`low >= high`) or a mismatch is found.

LISTING 9.1 CheckPalindrome.java

```
1 import java.util.Scanner;
2
3 public class CheckPalindrome {
4   /** Main method */
```

```
 5   public static void main(String[] args) {
 6     // Create a Scanner
 7     Scanner input = new Scanner(System.in);
 8
 9     // Prompt the user to enter a string
10     System.out.print("Enter a string: ");
11     String s = input.nextLine();                                  input string
12
13     if (isPalindrome(s))
14       System.out.println(s + " is a palindrome");
15     else
16       System.out.println(s + " is not a palindrome");
17   }
18
19   /** Check if a string is a palindrome */
20   public static boolean isPalindrome(String s) {
21     // The index of the first character in the string
22     int low = 0;                                                  low index
23
24     // The index of the last character in the string
25     int high = s.length() - 1;                                    high index
26
27     while (low < high) {
28       if (s.charAt(low) != s.charAt(high))
29         return false; // Not a palindrome
30
31       low++;                                                      update indices
32       high--;
33     }
34
35     return true; // The string is a palindrome
36   }
37 }
```

```
Enter a string: noon ↵Enter
noon is a palindrome

Enter a string: moon ↵Enter
moon is not a palindrome
```



The **nextLine()** method in the **Scanner** class (line 11) reads a line into **s**. **isPalindrome(s)** checks whether **s** is a palindrome (line 13).

9.2.13 Problem: Converting Hexadecimals to Decimals

Section 5.7 gives a program that converts a decimal to a hexadecimal. This section presents a program that converts a hex number into a decimal.

Given a hexadecimal number $h_n h_{n-1} h_{n-2} \ldots h_2 h_1 h_0$, the equivalent decimal value is

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

For example, the hex number **AB8C** is

$$10 \times 16^3 + 11 \times 16^2 + 8 \times 16^1 + 12 \times 16^0 = 43916$$

Our program will prompt the user to enter a hex number as a string and convert it into a decimal using the following method:

```
public static int hexToDecimal(String hex)
```

A brute-force approach is to convert each hex character into a decimal number, multiply it by 16^i for a hex digit at the `i`'s position, and add all the items together to obtain the equivalent decimal value for the hex number.

Note that

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_1 \times 16^1 + h_0 \times 16^0$$
$$= (\ldots((h_n \times 16 + h_{n-1}) \times 16 + h_{n-2}) \times 16 + \ldots + h_1) \times 16 + h_0$$

This observation leads to the following efficient algorithm for converting a hex string to a decimal number:

```
int decimalValue = 0;
for (int i = 0; i < hex.length(); i++) {
  char hexChar = hex.charAt(i);
  decimalValue = decimalValue * 16 + hexCharToDecimal(hexChar);
}
```

Here is a trace of the algorithm for hex number `AB8C`:

	i	hexChar	hexCharToDecimal(hexChar)	decimalValue
before the loop				0
after the 1st iteration	0	A	10	10
after the 2nd iteration	1	B	11	10 * 16 + 11
after the 3rd iteration	2	8	8	(10 * 16 + 11) * 16 + 8
after the 4th iteration	3	C	12	((10 * 16 + 11) * 16 + 8) * 16 + 12

Listing 9.2 gives the complete program.

LISTING 9.2 HexToDecimalConversion.java

```
1  import java.util.Scanner;
2
3  public class HexToDecimalConversion {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8
9      // Prompt the user to enter a string
10     System.out.print("Enter a hex number: ");
11     String hex = input.nextLine();
12
13     System.out.println("The decimal value for hex number "
14       + hex + " is " + hexToDecimal(hex.toUpperCase()));
15   }
16
17   public static int hexToDecimal(String hex) {
18     int decimalValue = 0;
19     for (int i = 0; i < hex.length(); i++) {
20       char hexChar = hex.charAt(i);
21       decimalValue = decimalValue * 16 + hexCharToDecimal(hexChar);
22     }
23
```

input string

hex to decimal

```
24      return decimalValue;
25    }
26
27    public static int hexCharToDecimal(char ch) {
28      if (ch >= 'A' && ch <= 'F')
29        return 10 + ch - 'A';
30      else // ch is '0', '1', ..., or '9'
31        return ch - '0';
32    }
33  }
```

hex char to decimal

```
Enter a hex number: AB8C Enter
The decimal value for hex number AB8C is 43916
```



```
Enter a hex number: af71 Enter
The decimal value for hex number af71 is 44913
```



The program reads a string from the console (line 11), and invokes the `hexToDecimal` method to convert a hex string to decimal number (line 14). The characters can be in either lowercase or uppercase. They are converted to uppercase before invoking the `hexToDecimal` method (line 14).

The `hexToDecimal` method is defined in lines 17–25 to return an integer. The length of the string is determined by invoking `hex.length()` in line 19.

The `hexCharToDecimal` method is defined in lines 27–32 to return a decimal value for a hex character. The character can be in either lowercase or uppercase. Recall that to subtract two characters is to subtract their Unicodes. For example, `'5' - '0'` is `5`.

9.3 The Character Class

Java provides a wrapper class for every primitive data type. These classes are `Character`, `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double` for `char`, `boolean`, `byte`, `short`, `int`, `long`, `float`, and `double`. All these classes are in the `java.lang` package. They enable the primitive data values to be treated as objects. They also contain useful methods for processing primitive values. This section introduces the `Character` class. The other wrapper classes will be introduced in Chapter 14, "Abstract Classes and Interfaces."

The `Character` class has a constructor and several methods for determining a character's category (uppercase, lowercase, digit, and so on) and for converting characters from uppercase to lowercase, and vice versa, as shown in Figure 9.10.

You can create a `Character` object from a `char` value. For example, the following statement creates a `Character` object for the character `'a'`.

```
Character character = new Character('a');
```

The `charValue` method returns the character value wrapped in the `Character` object. The `compareTo` method compares this character with another character and returns an integer that is the difference between the Unicodes of this character and the other character. The `equals` method returns `true` if and only if the two characters are the same. For example, suppose `charObject` is `new Character('b')`:

```
charObject.compareTo(new Character('a')) returns 1
charObject.compareTo(new Character('b')) returns 0
charObject.compareTo(new Character('c')) returns -1
charObject.compareTo(new Character('d')) returns -2
```

java.lang.Character	
+Character(value: char)	Constructs a character object with char value.
+charValue(): char	Returns the char value from this object.
+compareTo(anotherCharacter: Character): int	Compares this character with another.
+equals(anotherCharacter: Character): boolean	Returns true if this character is equal to another.
+isDigit(ch: char): boolean	Returns true if the specified character is a digit.
+isLetter(ch: char): boolean	Returns true if the specified character is a letter.
+isLetterOrDigit(ch: char): boolean	Returns true if the character is a letter or a digit.
+isLowerCase(ch: char): boolean	Returns true if the character is a lowercase letter.
+isUpperCase(ch: char): boolean	Returns true if the character is an uppercase letter.
+toLowerCase(ch: char): char	Returns the lowercase of the specified character.
+toUpperCase(ch: char): char	Returns the uppercase of the specified character.

FIGURE 9.10 The `Character` class provides the methods for manipulating a character.

```
charObject.equals(new Character('b')) returns true
charObject.equals(new Character('d')) returns false
```

Most of the methods in the `Character` class are static methods. The `isDigit(char ch)` method returns `true` if the character is a digit. The `isLetter(char ch)` method returns `true` if the character is a letter. The `isLetterOrDigit(char ch)` method returns `true` if the character is a letter or a digit. The `isLowerCase(char ch)` method returns `true` if the character is a lowercase letter. The `isUpperCase(char ch)` method returns `true` if the character is an uppercase letter. The `toLowerCase(char ch)` method returns the lowercase letter for the character, and the `toUpperCase(char ch)` method returns the uppercase letter for the character.

9.3.1 Problem: Counting Each Letter in a String

The problem is to write a program that prompts the user to enter a string and counts the number of occurrences of each letter in the string regardless of case.

Here are the steps to solve this problem:

1. Convert all the uppercase letters in the string to lowercase using the `toLowerCase` method in the `String` class.
2. Create an array, say `counts` of `26 int` values, each of which counts the occurrences of a letter. That is, `counts[0]` counts the number of `a`'s, `counts[1]` counts the number of `b`'s, and so on.
3. For each character in the string, check whether it is a (lowercase) letter. If so, increment the corresponding count in the array.

Listing 9.3 gives the complete program:

LISTING 9.3 CountEachLetter.java

```
1 import java.util.Scanner;
2
3 public class CountEachLetter {
4   /** Main method */
5   public static void main(String[] args) {
6     // Create a Scanner
7     Scanner input = new Scanner(System.in);
```

```
 8
 9     // Prompt the user to enter a string
10     System.out.print("Enter a string: ");
11     String s = input.nextLine();                                  input string
12
13     // Invoke the countLetters method to count each letter
14     int[] counts = countLetters(s.toLowerCase()) ;                count letters
15
16     // Display results
17     for (int i = 0; i < counts.length; i++) {
18       if (counts[i] != 0)
19         System.out.println((char)('a' + i) + " appears  " +
20           counts[i] + ((counts[i] == 1) ? " time" : " times"));
21     }
22   }
23
24   /** Count each letter in the string */
25   public static int[] countLetters(String s ) {
26     int[] counts = new int[26];
27
28     for (int i = 0; i < s.length() ; i++) {
29       if (Character.isLetter(s.charAt(i)) )
30         counts[s.charAt(i) - 'a']++;                              count a letter
31     }
32
33     return counts;
34   }
35 }
```

```
Enter a string: ababab x  Enter
a appears  3 times
b appears  3 times
x appears  1 time
```



The main method reads a line (line 11) and counts the number of occurrences of each letter in the string by invoking the `countLetters` method (line 14). Since the case of the letters is ignored, the program uses the `toLowerCase` method to convert the string into all lowercase and pass the new string to the `countLetters` method.

The `countLetters` method (lines 25–34) returns an array of `26` elements. Each element counts the number of occurrences of a letter in the string `s`. The method processes each character in the string. If the character is a letter, its corresponding count is increased by `1`. For example, if the character (`s.charAr(i)`) is `'a'`, the corresponding count is `counts['a' - 'a']` (i.e., `counts[0]`). If the character is `'b'`, the corresponding count is `counts['b' - 'a']` (i.e., `counts[1]`), since the Unicode of `'b'` is `1` more than that of `'a'`. If the character is `'z'`, the corresponding count is `counts['z' - 'a']` (i.e., `counts[25]`), since the Unicode of `'z'` is `25` more than that of `'a'`.

9.4 The StringBuilder/StringBuffer Class

The `StringBuilder`/`StringBuffer` class is an alternative to the `String` class. In general, a `StringBuilder`/`StringBuffer` can be used wherever a string is used. `StringBuilder`/`StringBuffer` is more flexible than `String`. You can add, insert, or append new contents into a `StringBuilder` or a `StringBuffer`, whereas the value of a `String` object is fixed, once the string is created.

StringBuilder

The `StringBuilder` class is similar to `StringBuffer` except that the methods for modifying buffer in `StringBuffer` are synchronized. Use `StringBuffer` if it may be accessed by multiple tasks concurrently. Using `StringBuilder` is more efficient if it is accessed by a single task. The constructors and methods in `StringBuffer` and `StringBuilder` are almost the same. This section covers `StringBuilder`. You may replace `StringBuilder` by `StringBuffer`. The program can compile and run without any other changes.

StringBuilder constructors

The `StringBuilder` class has three constructors and more than 30 methods for managing the builder and modifying strings in the builder. You can create an empty string builder or a string builder from a string using the constructors, as shown in Figure 9.11.

java.lang.StringBuilder	
`+StringBuilder()`	Constructs an empty string builder with capacity 16.
`+StringBuilder(capacity: int)`	Constructs a string builder with the specified capacity.
`+StringBuilder(s: String)`	Constructs a string builder with the specified string.

FIGURE 9.11 The `StringBuilder` class contains the constructors for creating instances of `StringBuilder`.

9.4.1 Modifying Strings in the `StringBuilder`

You can append new contents at the end of a string builder, insert new contents at a specified position in a string builder, and delete or replace characters in a string builder, using the methods listed in Figure 9.12:

java.lang.StringBuilder	
`+append(data: char[]): StringBuilder`	Appends a `char` array into this string builder.
`+append(data: char[], offset: int, len: int): StringBuilder`	Appends a subarray in `data` into this string builder.
`+append(v: aPrimitiveType): StringBuilder`	Appends a primitive type value as a string to this builder.
`+append(s: String): StringBuilder`	Appends a string to this string builder.
`+delete(startIndex: int, endIndex: int): StringBuilder`	Deletes characters from `startIndex` to `endIndex-1`.
`+deleteCharAt(index: int): StringBuilder`	Deletes a character at the specified index.
`+insert(index: int, data: char[], offset: int, len: int): StringBuilder`	Inserts a subarray of the data in the array to the builder at the specified index.
`+insert(offset: int, data: char[]): StringBuilder`	Inserts data into this builder at the position offset.
`+insert(offset: int, b: aPrimitiveType): StringBuilder`	Inserts a value converted to a string into this builder.
`+insert(offset: int, s: String): StringBuilder`	Inserts a string into this builder at the position offset.
`+replace(startIndex: int, endIndex: int, s: String): StringBuilder`	Replaces the characters in this builder from `startIndex` to `endIndex-1` with the specified string.
`+reverse(): StringBuilder`	Reverses the characters in the builder.
`+setCharAt(index: int, ch: char): void`	Sets a new character at the specified index in this builder.

FIGURE 9.12 The `StringBuilder` class contains the methods for modifying string builders.

The `StringBuilder` class provides several overloaded methods to append `boolean`, `char`, `char array`, `double`, `float`, `int`, `long`, and `String` into a string builder. For example, the following code appends strings and characters into `stringBuilder` to form a new string, `"Welcome to Java"`.

```
StringBuilder stringBuilder = new StringBuilder();
```

```
stringBuilder.append("Welcome");
stringBuilder.append(' ');
stringBuilder.append("to");
stringBuilder.append(' ');
stringBuilder.append("Java");
```

append

The `StringBuilder` class also contains overloaded methods to insert `boolean`, `char`, `char array`, `double`, `float`, `int`, `long`, and `String` into a string builder. Consider the following code:

```
stringBuilder.insert(11, "HTML and ");
```

insert

Suppose `stringBuilder` contains `"Welcome to Java"` before the `insert` method is applied. This code inserts `"HTML and "` at position 11 in `stringBuilder` (just before `J`). The new `stringBuilder` is `"Welcome to HTML and Java"`.

You can also delete characters from a string in the builder using the two `delete` methods, reverse the string using the `reverse` method, replace characters using the `replace` method, or set a new character in a string using the `setCharAt` method.

For example, suppose `stringBuilder` contains `"Welcome to Java"` before each of the following methods is applied.

`stringBuilder.delete(8, 11)` changes the builder to `Welcome Java`. **delete**
`stringBuilder.deleteCharAt(8)` changes the builder to `Welcome o Java`. **deleteCharAt**
`stringBuilder.reverse()` changes the builder to `avaJ ot emocleW`. **reverse**
`stringBuilder.replace(11, 15, "HTML")` changes the builder to `Welcome to HTML`. **replace**
`stringBuilder.setCharAt(0, 'w')` sets the builder to `welcome to Java`. **setCharAt**

All these modification methods except `setCharAt` do two things:

1. Change the contents of the string builder
2. Return the reference of the string builder

For example, the following statement

```
StringBuilder stringBuilder1 = stringBuilder.reverse();
```

reverses the string in the builder and assigns the reference of the builder to `stringBuilder1`. Thus, `stringBuilder` and `stringBuilder1` both point to the same `StringBuilder` object. Recall that a value-returning method may be invoked as a statement, if you are not interested in the return value of the method. In this case, the return value is simply ignored. For example, in the following statement

ignore return value

```
stringBuilder.reverse();
```

the return value is ignored.



Tip

If a string does not require any change, use `String` rather than `StringBuilder`. Java can perform some optimizations for `String`, such as sharing interned strings.

String or **StringBuilder**?

9.4.2 The `toString`, `capacity`, `length`, `setLength`, and `charAt` Methods

The `StringBuilder` class provides the additional methods for manipulating a string builder and obtaining its properties, as shown in Figure 9.13.

java.lang.StringBuilder	
+toString(): String	Returns a string object from the string builder.
+capacity(): int	Returns the capacity of this string builder.
+charAt(index: int): char	Returns the character at the specified index.
+length(): int	Returns the number of characters in this builder.
+setLength(newLength: int): void	Sets a new length in this builder.
+substring(startIndex: int): String	Returns a substring starting at `startIndex`.
+substring(startIndex: int, endIndex: int): String	Returns a substring from `startIndex` to `endIndex-1`.
+trimToSize(): void	Reduces the storage size used for the string builder.

FIGURE 9.13 The `StringBuilder` class contains the methods for modifying string builders.

`capacity()`

The `capacity()` method returns the current capacity of the string builder. The capacity is the number of characters it is able to store without having to increase its size.

`length()`
`setLength(int)`

The `length()` method returns the number of characters actually stored in the string builder. The `setLength(newLength)` method sets the length of the string builder. If the `newLength` argument is less than the current length of the string builder, the string builder is truncated to contain exactly the number of characters given by the `newLength` argument. If the `newLength` argument is greater than or equal to the current length, sufficient null characters (`'\u0000'`) are appended to the string builder so that `length` becomes the `newLength` argument. The `newLength` argument must be greater than or equal to `0`.

`charAt(int)`

The `charAt(index)` method returns the character at a specific `index` in the string builder. The index is `0` based. The first character of a string builder is at index `0`, the next at index `1`, and so on. The `index` argument must be greater than or equal to `0`, and less than the length of the string builder.



Note

length and capacity

The length of the string is always less than or equal to the capacity of the builder. The length is the actual size of the string stored in the builder, and the capacity is the current size of the builder. The builder's capacity is automatically increased if more characters are added to exceed its capacity. Internally, a string builder is an array of characters, so the builder's capacity is the size of the array. If the builder's capacity is exceeded, the array is replaced by a new array. The new array size is `2 * (the previous array size + 1)`.



Tip

initial capacity

You can use `new StringBuilder(initialCapacity)` to create a `StringBuilder` with a specified initial capacity. By carefully choosing the initial capacity, you can make your program more efficient. If the capacity is always larger than the actual length of the builder, the JVM will never need to reallocate memory for the builder. On the other hand, if the capacity is too large, you will waste memory space. You can use the `trimToSize()` method to reduce the capacity to the actual size.

`trimToSize()`

9.4.3 Problem: Ignoring Nonalphanumeric Characters When Checking Palindromes

Listing 9.1, CheckPalindrome.java, considered all the characters in a string to check whether it was a palindrome. Write a new program that ignores nonalphanumeric characters in checking whether a string is a palindrome.

Here are the steps to solve the problem:

1. Filter the string by removing the nonalphanumeric characters. This can be done by creating an empty string builder, adding each alphanumeric character in the string to a string builder, and returning the string from the string builder. You can use the

`isLetterOrDigit(ch)` method in the `Character` class to check whether character `ch` is a letter or a digit.

2. Obtain a new string that is the reversal of the filtered string. Compare the reversed string with the filtered string using the `equals` method.

The complete program is shown in Listing 9.4.

LISTING 9.4 PalindromeIgnoreNonAlphanumeric.java

```
1  import java.util.Scanner;
2
3  public class PalindromeIgnoreNonAlphanumeric {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8
9      // Prompt the user to enter a string
10     System.out.print("Enter a string: ");
11     String s = input.nextLine();
12
13     // Display result
14     System.out.println("Ignoring nonalphanumeric characters, \nis "
15       + s + " a palindrome? " + isPalindrome(s));
16   }
17
18   /** Return true if a string is a palindrome */
19   public static boolean isPalindrome(String s) {                    check palindrome
20     // Create a new string by eliminating nonalphanumeric chars
21     String s1 = filter(s);
22
23     // Create a new string that is the reversal of s1
24     String s2 = reverse(s1);
25
26     // Compare if the reversal is the same as the original string
27     return s2.equals(s1);
28   }
29
30   /** Create a new string by eliminating nonalphanumeric chars */
31   public static String filter(String s) {
32     // Create a string builder
33     StringBuilder stringBuilder = new StringBuilder();
34
35     // Examine each char in the string to skip alphanumeric char
36     for (int i = 0; i < s.length(); i++) {
37       if (Character.isLetterOrDigit(s.charAt(i))) {
38         stringBuilder.append(s.charAt(i));                          add letter or digit
39       }
40     }
41
42     // Return a new filtered string
43     return stringBuilder.toString();
44   }
45
46   /** Create a new string by reversing a specified string */
47   public static String reverse(String s) {
48     StringBuilder stringBuilder = new StringBuilder(s);
49     stringBuilder.reverse(); // Invoke reverse in StringBuilder
50     return stringBuilder.toString();
51   }
52 }
```



```
Enter a string: ab<c>cb?a ↵Enter
Ignoring nonalphanumeric characters,
is ab<c>cb?a a palindrome? true

Enter a string: abcc><?cab ↵Enter
Ignoring nonalphanumeric characters,
is abcc><?cab a palindrome? false
```

The `filter(String s)` method (lines 31–44) examines each character in string `s` and copies it to a string builder if the character is a letter or a numeric character. The `filter` method returns the string in the builder. The `reverse(String s)` method (lines 47–52) creates a new string that reverses the specified string `s`. The `filter` and `reverse` methods both return a new string. The original string is not changed.

The program in Listing 9.1 checks whether a string is a palindrome by comparing pairs of characters from both ends of the string. Listing 9.4 uses the `reverse` method in the `StringBuilder` class to reverse the string, then compares whether the two strings are equal to determine whether the original string is a palindrome.

9.5 Command-Line Arguments

Perhaps you have already noticed the unusual declarations for the `main` method, which has parameter `args` of `String[]` type. It is clear that `args` is an array of strings. The `main` method is just like a regular method with a parameter. You can call a regular method by passing actual parameters. Can you pass arguments to `main`? Yes, of course you can. For example, the `main` method in class `TestMain` is invoked by a method in `A`, as shown below:

```
public class A {
  public static void main(String[] args) {
    String[] strings = {"New York",
      "Boston", "Atlanta"};
    TestMain.main(strings);
  }
}
```

```
public class TestMain {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

A main method is just a regular method. Furthermore, you can pass arguments from the command line.

9.5.1 Passing Strings to the `main` Method

You can pass strings to a `main` method from the command line when you run the program. The following command line, for example, starts the program `TestMain` with three strings: `arg0`, `arg1`, and `arg2`:

```
java TestMain arg0 arg1 arg2
```

`arg0`, `arg1`, and `arg2` are strings, but they don't have to appear in double quotes on the command line. The strings are separated by a space. A string that contains a space must be enclosed in double quotes. Consider the following command line:

```
java TestMain "First num" alpha 53
```

It starts the program with three strings: `"First num"`, `alpha`, and `53`, a numeric string. Since `"First num"` is a string, it is enclosed in double quotes. Note that `53` is actually treated as a string. You can use `"53"` instead of `53` in the command line.

When the `main` method is invoked, the Java interpreter creates an array to hold the command-line arguments and pass the array reference to `args`. For example, if you invoke a program with `n` arguments, the Java interpreter creates an array like this one:

```
args = new String[n];
```

The Java interpreter then passes `args` to invoke the `main` method.



Note

If you run the program with no strings passed, the array is created with `new String[0]`. In this case, the array is empty with length `0`. `args` references to this empty array. Therefore, `args` is not `null`, but `args.length` is `0`.

9.5.2 Problem: Calculator



Video Note
Command-line argument

Suppose you are to develop a program that performs arithmetic operations on integers. The program receives three arguments: an integer followed by an operator and another integer. For example, to add two integers, use this command:

```
java Calculator 2 + 3
```

The program will display the following output:

```
2 + 3 = 5
```

Figure 9.14 shows sample runs of the program.

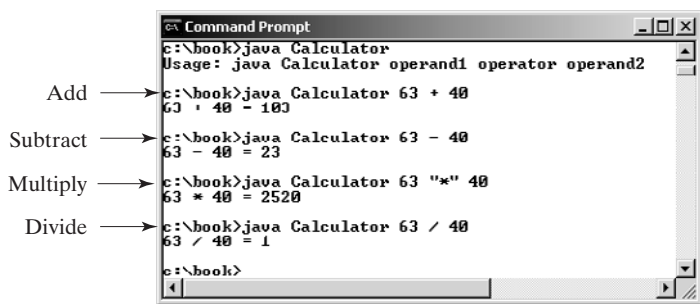


FIGURE 9.14 The program takes three arguments (operand1 operator operand2) from the command line and displays the expression and the result of the arithmetic operation.

The strings passed to the main program are stored in `args`, which is an array of strings. The first string is stored in `args[0]`, and `args.length` is the number of strings passed.

Here are the steps in the program:

- Use `args.length` to determine whether three arguments have been provided in the command line. If not, terminate the program using `System.exit(0)`.
- Perform a binary arithmetic operation on the operands `args[0]` and `args[2]` using the operator specified in `args[1]`.

The program is shown in Listing 9.5.

LISTING 9.5 Calculator.java

```
1 public class Calculator {
2   /** Main method */
3   public static void main(String[] args) {
```

```
4     // Check number of strings passed
5     if (args.length != 3) {
6       System.out.println(
7         "Usage: java Calculator operand1 operator operand2");
8       System.exit(0);
9     }
10
11    // The result of the operation
12    int result = 0;
13
14    // Determine the operator
15    switch (args[1].charAt(0)) {
16      case '+': result = Integer.parseInt(args[0]) +
17                         Integer.parseInt(args[2]);
18                break;
19      case '-': result = Integer.parseInt(args[0]) -
20                         Integer.parseInt(args[2]);
21                break;
22      case '*': result = Integer.parseInt(args[0]) *
23                         Integer.parseInt(args[2]);
24                break;
25      case '/': result = Integer.parseInt(args[0]) /
26                         Integer.parseInt(args[2]);
27    }
28
29    // Display result
30    System.out.println(args[0] + ' ' + args[1] + ' ' + args[2]
31      + " = " + result);
32  }
33 }
```

check operator

`Integer.parseInt(args[0])` (line 16) converts a digital string into an integer. The string must consist of digits. If not, the program will terminate abnormally.



Note

special * character

In the sample run, `"*"` had to be used instead of * for the command

```
java Calculator 63 "*" 40
```

The * symbol refers to all the files in the current directory when it is used on a command line. Therefore, in order to specify the multiplication operator, the * must be enclosed in quote marks in the command line. The following program displays all the files in the current directory when issuing the command `java Test *`:

```
public class Test {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

9.6 The `File` Class

why file?

Data stored in variables, arrays, and objects are temporary; they are lost when the program terminates. To permanently store the data created in a program, you need to save them in a file on a disk or a CD. The file can be transported and can be read later by other programs. Since data are stored in files, this section introduces how to use the `File` class to obtain file properties and to delete and rename files. The next section introduces how to read/write data from/to text files.

Every file is placed in a directory in the file system. An *absolute file name* contains a file name with its complete path and drive letter. For example, **c:\book\Welcome.java** is the absolute file name for the file **Welcome.java** on the Windows operating system. Here **c:\book** is referred to as the *directory path* for the file. Absolute file names are machine dependent. On the Unix platform, the absolute file name may be **/home/liang/book/Welcome.java**, where **/home/liang/book** is the directory path for the file **Welcome.java**.

absolute file name

directory path

The `File` class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion. The `File` class contains the methods for obtaining file properties and for renaming and deleting files, as shown in Figure 9.15. However, *the `File` class does not contain the methods for reading and writing file contents*.

java.io.File	
`+File(pathname: String)`	Creates a `File` object for the specified path name. The path name may be a directory or a file.
`+File(parent: String, child: String)`	Creates a `File` object for the child under the directory parent. The child may be a file name or a subdirectory.
`+File(parent: File, child: String)`	Creates a `File` object for the child under the directory parent. The parent is a `File` object. In the preceding constructor, the parent is a string.
`+exists(): boolean`	Returns true if the file or the directory represented by the `File` object exists.
`+canRead(): boolean`	Returns true if the file represented by the `File` object exists and can be read.
`+canWrite(): boolean`	Returns true if the file represented by the `File` object exists and can be written.
`+isDirectory(): boolean`	Returns true if the `File` object represents a directory.
`+isFile(): boolean`	Returns true if the `File` object represents a file.
`+isAbsolute(): boolean`	Returns true if the `File` object is created using an absolute path name.
`+isHidden(): boolean`	Returns true if the file represented in the `File` object is hidden. The exact definition of *hidden* is system dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period character '.'.
`+getAbsolutePath(): String`	Returns the complete absolute file or directory name represented by the `File` object.
`+getCanonicalPath(): String`	Returns the same as `getAbsolutePath()` except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix platforms), and converts drive letters to standard uppercase (on Win32 platforms).
`+getName(): String`	Returns the last name of the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getName()` returns `test.dat`.
`+getPath(): String`	Returns the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getPath()` returns `c:\book\test.dat`.
`+getParent(): String`	Returns the complete parent directory of the current directory or the file represented by the `File` object. For example, new `File("c:\\book\\test.dat").getParent()` returns `c:\book`.
`+lastModified(): long`	Returns the time that the file was last modified.
`+length(): long`	Returns the size of the file, or 0 if it does not exist or if it is a directory.
`+listFile(): File[]`	Returns the files under the directory for a directory `File` object.
`+delete(): boolean`	Deletes this file. The method returns true if the deletion succeeds.
`+renameTo(dest: File): boolean`	Renames this file. The method returns true if the operation succeeds.

FIGURE 9.15 The `File` class can be used to obtain file and directory properties and to delete and rename files.

The file name is a string. The `File` class is a wrapper class for the file name and its directory path. For example, `new File("c:\\book")` creates a `File` object for the directory **c:\book**, and `new File("c:\\book\\test.dat")` creates a `File` object for the file **c:\\book\\test.dat**, both on Windows. You can use the `File` class's `isDirectory()` method to check whether the object represents a directory, and the `isFile()` method to check whether the object represents a file.



Caution

The directory separator for Windows is a backslash (\). The backslash is a special character in Java and should be written as \\ in a string literal (see Table 2.6).

\ in file names



Note

Constructing a `File` *instance does not create a file on the machine.* You can create a `File` instance for any file name regardless whether it exists or not. You can invoke the `exists()` method on a `File` instance to check whether the file exists.

relative file name

Java directory separator (`/`)

Do not use absolute file names in your program. If you use a file name such as `"c:\\book\\Welcome.java"`, it will work on Windows but not on other platforms. You should use a file name relative to the current directory. For example, you may create a `File` object using `new File("Welcome.java")` for the file **Welcome.java** in the current directory. You may create a `File` object using `new File("image/us.gif")` for the file **us.gif** under the **image** directory in the current directory. The forward slash (`/`) is the Java directory separator, which is the same as on Unix. The statement `new File("image/us.gif")` works on Windows, Unix, and any other platform.

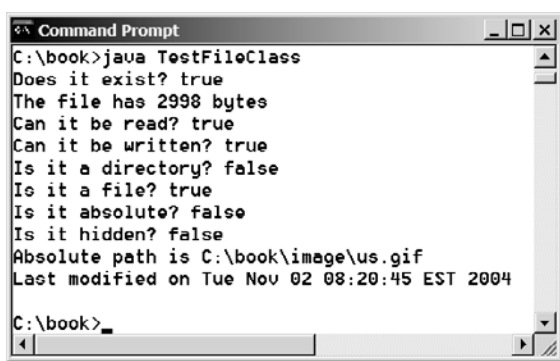
Listing 9.6 demonstrates how to create a `File` object and use the methods in the `File` class to obtain its properties. The program creates a `File` object for the file **us.gif**. This file is stored under the **image** directory in the current directory.

LISTING 9.6 TestFileClass.java

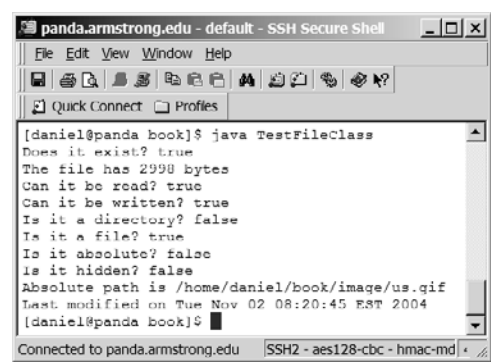
create a `File`
`exists()`
`length()`
`canRead()`
`canWrite()`
`isDirectory()`
`isFile()`
`isAbsolute()`
`isHidden()`
`getAbsolutePath()`
`lastModified()`

```
1  public class TestFileClass {
2    public static void main(String[] args) {
3      java.io.File file = new java.io.File("image/us.gif");
4      System.out.println("Does it exist? " + file.exists());
5      System.out.println("The file has " + file.length() + " bytes");
6      System.out.println("Can it be read? " + file.canRead());
7      System.out.println("Can it be written? " + file.canWrite());
8      System.out.println("Is it a directory? " + file.isDirectory());
9      System.out.println("Is it a file? " + file.isFile());
10     System.out.println("Is it absolute? " + file.isAbsolute());
11     System.out.println("Is it hidden? " + file.isHidden());
12     System.out.println("Absolute path is " +
13       file.getAbsolutePath());
14     System.out.println("Last modified on " +
15       new java.util.Date(file.lastModified()));
16   }
17 }
```

The `lastModified()` method returns the date and time when the file was last modified, measured in milliseconds since the beginning of Unix time (00:00:00 GMT, January 1, 1970). The `Date` class is used to display it in a readable format in lines 14–15.


Command Prompt
C:\book>java TestFileClass
Does it exist? true
The file has 2998 bytes
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
Absolute path is C:\book\image\us.gif
Last modified on Tue Nov 02 08:20:45 EST 2004
C:\book>

(a) On Windows


panda.armstrong.edu - default - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles
[daniel@panda book]$ java TestFileClass
Does it exist? true
The file has 2998 bytes
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
Absolute path is /home/daniel/book/image/us.gif
Last modified on Tue Nov 02 08:20:45 EST 2004
[daniel@panda book]$
Connected to panda.armstrong.edu
SSH2 - aes128-cbc - hmac-md

(b) On Unix

FIGURE 9.16 The program creates a `File` object and displays file properties.

Figure 9.16(a) shows a sample run of the program on Windows, and Figure 9.16(b), a sample run on Unix. As shown in the figures, the path-naming conventions on Windows are different from those on Unix.

9.7 File Input and Output

A `File` object encapsulates the properties of a file or a path but does not contain the methods for creating a file or for reading/writing data from/to a file. In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file. This section introduces how to read/write strings and numeric values from/to a text file using the `Scanner` and `PrintWriter` classes.

9.7.1 Writing Data Using `PrintWriter`

The `java.io.PrintWriter` class can be used to create a file and write data to a text file. First, you have to create a `PrintWriter` object for a text file as follows:

```
PrintWriter output = new PrintWriter(filename);
```

Then, you can invoke the `print`, `println`, and `printf` methods on the `PrintWriter` object to write data to a file. Figure 9.17 summarizes frequently used methods in `PrintWriter`.

java.io.PrintWriter	
+PrintWriter(file: File)	Creates a PrintWriter object for the specified file object.
+PrintWriter(filename: String)	Creates a PrintWriter object for the specified file-name string.
+print(s: String): void	Writes a string to the file.
+print(c: char): void	Writes a character to the file.
+print(cArray: char[]): void	Writes an array of characters to the file.
+print(i: int): void	Writes an int value to the file.
+print(l: long): void	Writes a long value to the file.
+print(f: float): void	Writes a float value to the file.
+print(d: double): void	Writes a double value to the file.
+print(b: boolean): void	Writes a boolean value to the file.
Also contains the overloaded println methods.	A println method acts like a print method; additionally it prints a line separator. The line-separator string is defined by the system. It is \r\n on Windows and \n on Unix.
Also contains the overloaded printf methods.	The printf method was introduced in §3.17, "Formatting Console Output."

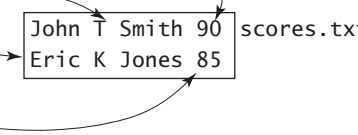
FIGURE 9.17 The `PrintWriter` class contains the methods for writing data to a text file.

Listing 9.7 gives an example that creates an instance of `PrintWriter` and writes two lines to the file "scores.txt". Each line consists of first name (a string), middle-name initial (a character), last name (a string), and score (an integer).

LISTING 9.7 `WriteData.java`

```
1  public class WriteData {
2    public static void main(String[] args) throws Exception {          throws an exception
3      java.io.File file = new java.io.File("scores.txt");              create File object
4      if (file.exists()) {                                             file exist?
5        System.out.println("File already exists");
6        System.exit(0);
7      }
8
9      // Create a file
10     java.io.PrintWriter output = new java.io.PrintWriter(file);      create PrintWriter
```

```
11
12      // Write formatted output to the file
13      output.print("John T Smith ");
14      output.println(90);
15      output.print("Eric K Jones ");
16      output.println(85);
17
18      // Close the file
19      output.close();
20    }
21  }
```



print data

close file

Lines 3–7 check whether the file scores.txt exists. If so, exit the program (line 6).

create a file

Invoking the constructor of **`PrintWriter`** will create a new file if the file does not exist. If the file already exists, the current content in the file will be discarded.

Invoking the constructor of **`PrintWriter`** may throw an I/O exception. Java forces you to write the code to deal with this type of exception. You will learn how to handle it in Chapter 13, "Exception Handling." For now, simply declare ***`throws Exception`*** in the method header (line 2).

`throws Exception`

You have used the **`System.out.print`** and **`System.out.println`** methods to write text to the console. **`System.out`** is a standard Java object for the console. You can create objects for writing text to any file using **`print`**, **`println`**, and **`printf`** (lines 13–16).

`print` method

close file

The **`close()`** method must be used to close the file. If this method is not invoked, the data may not be saved properly in the file.

9.7.2 Reading Data Using `Scanner`

The **`java.util.Scanner`** class was used to read strings and primitive values from the console in §2.3, "Reading Input from the Console." A **`Scanner`** breaks its input into tokens delimited by whitespace characters. To read from the keyboard, you create a **`Scanner`** for **`System.in`**, as follows:

```
Scanner input = new Scanner(System.in);
```

To read from a file, create a **`Scanner`** for a file, as follows:

```
Scanner input = new Scanner(new File(filename));
```

Figure 9.18 summarizes frequently used methods in **`Scanner`**.

java.util.Scanner

`+Scanner(source: File)`	Creates a scanner that produces values scanned from the specified file.
`+Scanner(source: String)`	Creates a scanner that produces values scanned from the specified string.
`+close()`	Closes this scanner.
`+hasNext(): boolean`	Returns true if this scanner has more data to be read.
`+next(): String`	Returns next token as a string from this scanner.
`+nextLine(): String`	Returns a line ending with the line separator from this scanner.
`+nextByte(): byte`	Returns next token as a `byte` from this scanner.
`+nextShort(): short`	Returns next token as a `short` from this scanner.
`+nextInt(): int`	Returns next token as an `int` from this scanner.
`+nextLong(): long`	Returns next token as a `long` from this scanner.
`+nextFloat(): float`	Returns next token as a `float` from this scanner.
`+nextDouble(): double`	Returns next token as a `double` from this scanner.
`+useDelimiter(pattern: String): Scanner`	Sets this scanner's delimiting pattern and returns this scanner.

FIGURE 9.18 The **`Scanner`** class contains the methods for scanning data.

Listing 9.8 gives an example that creates an instance of `Scanner` and reads data from the file "scores.txt".

LISTING 9.8 ReadData.java

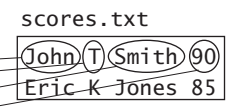
```
1 import java.util.Scanner;
2
3 public class ReadData {
4   public static void main(String[] args) throws Exception {
5     // Create a File instance
6     java.io.File file = new java.io.File("scores.txt");
7
8     // Create a Scanner for the file
9     Scanner input = new Scanner(file);
10
11    // Read data from a file
12    while (input.hasNext()) {
13      String firstName = input.next();
14      String mi = input.next();
15      String lastName = input.next();
16      int score = input.nextInt();
17      System.out.println(
18        firstName + " " + mi + " " + lastName + " " + score);
19    }
20
21    // Close the file
22    input.close();
23  }
24 }
```

create a **File**

create a **Scanner**

has next?
read items

close file



Note that `new Scanner(String)` creates a `Scanner` for a given string. To create a `Scanner` to read data from a file, you have to use the `java.io.File` class to create an instance of the `File` using the constructor `new File(filename)` (line 6), and use `new Scanner(File)` to create a `Scanner` for the file (line 9).

File class

Invoking the constructor `new Scanner(File)` may throw an I/O exception. So the `main` method declares `throws Exception` in line 4.

throws Exception

Each iteration in the `while` loop reads first name, mi, last name, and score from the text file (lines 12–19). The file is closed in line 22.

It is not necessary to close the input file (line 22), but it is a good practice to do so to release the resources occupied by the file.

close file

9.7.3 How Does Scanner Work?

The `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`, and `next()` methods are known as *token-reading methods*, because they read tokens separated by delimiters. By default, the delimiters are whitespace. You can use the `useDelimiter(String regex)` method to set a new pattern for delimiters.

token-reading method

change delimiter

How does an input method work? A token-reading method first skips any delimiters (whitespace by default), then reads a token ending at a delimiter. The token is then automatically converted into a value of the `byte`, `short`, `int`, `long`, `float`, or `double` type for `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, and `nextDouble()`, respectively. For the `next()` method, no conversion is performed. If the token does not match the expected type, a runtime exception `java.util.InputMismatchException` will be thrown.

InputMismatchException

Both methods `next()` and `nextLine()` read a string. The `next()` method reads a string delimited by delimiters, but `nextLine()` reads a line ending with a line separator.

next() vs. **nextLine()**

line separator



Note

The line-separator string is defined by the system. It is `\r\n` on Windows and `\n` on Unix. To get the line separator on a particular platform, use

```
String lineSeparator = System.getProperty("line.separator");
```

If you enter input from a keyboard, a line ends with the *Enter* key, which corresponds to the `\n` character.

behavior of `nextLine()`

The token-reading method does not read the delimiter after the token. If the `nextLine()` is invoked after a token-reading method, the method reads characters that start from this delimiter and end with the line separator. The line separator is read, but it is not part of the string returned by `nextLine()`.

input from file

Suppose a text file named test.txt contains a line

```
34 567
```

After the following code is executed,

```
Scanner input = new Scanner(new File("test.txt"));
int intValue = input.nextInt();
String line = input.nextLine();
```

`intValue` contains `34` and `line` contains characters `' '`, `'5'`, `'6'`, `'7'`.

input from keyboard

What happens if the input is *entered from the keyboard*? Suppose you enter `34`, the *Enter* key, `567`, and the *Enter* key for the following code:

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
String line = input.nextLine();
```

You will get `34` in `intValue` and an empty string in `line`. Why? Here is the reason. The token-reading method `nextInt()` reads in `34` and stops at the delimiter, which in this case is a line separator (the *Enter* key). The `nextLine()` method ends after reading the line separator and returns the string read before the line separator. Since there are no characters before the line separator, `line` is empty.

9.7.4 Problem: Replacing Text

Suppose you are to write a program named `ReplaceText` that replaces all occurrences of a string in a text file with a new string. The file name and strings are passed as command-line arguments as follows:

```
java ReplaceText sourceFile targetFile oldString newString
```

For example, invoking

```
java ReplaceText FormatString.java t.txt StringBuilder StringBuffer
```

replaces all the occurrences of `StringBuilder` by `StringBuffer` in FormatString.java and saves the new file in t.txt.

Listing 9.9 gives the solution to the problem. The program checks the number of arguments passed to the `main` method (lines 7–11), checks whether the source and target files exist (lines 14–25), creates a `Scanner` for the source file (line 28), creates a `PrintWriter` for the target file, and repeatedly reads a line from the source file (line 32), replaces the text (line 33), and writes a new line to the target file (line 34). You must close the output file (line 38) to ensure that data are saved to the file properly.

LISTING 9.9 ReplaceText.java

```
1 import java.io.*;
2 import java.util.*;
3
4 public class ReplaceText {
5   public static void main(String[] args) throws Exception {
6     // Check command-line parameter usage
7     if (args.length != 4) {                                          check command usage
8       System.out.println(
9         "Usage: java ReplaceText sourceFile targetFile oldStr newStr");
10      System.exit(0);
11    }
12
13    // Check if source file exists
14    File sourceFile = new File(args[0]);
15    if (!sourceFile.exists() ) {                                     source file exists?
16       System.out.println("Source file " + args[0] + " does not exist");
17       System.exit(0);
18    }
19
20    // Check if target file exists
21    File targetFile = new File(args[1]);
22    if (targetFile.exists() ) {                                      target file exists?
23      System.out.println("Target file " + args[1] + " already exists");
24      System.exit(0);
25    }
26
27    // Create a Scanner for input and a PrintWriter for output
28    Scanner input = new Scanner(sourceFile);                         create a Scanner
29    PrintWriter output = new PrintWriter(targetFile);                create a PrintWriter
30
31    while (input.hasNext() ) {                                       has next?
32      String s1 = input.nextLine();                                  read a line
33      String s2 = s1.replaceAll(args[2], args[3]);
34      output.println(s2);
35    }
36
37    input.close();                                                   close file
38    output.close();
39  }
40 }
```

9.8 (GUI) File Dialogs

Java provides the `javax.swing.JFileChooser` class for displaying a file dialog, as shown in Figure 9.19. From this dialog box, the user can choose a file.

Listing 9.10 gives a program that prompts the user to choose a file and displays its contents on the console.

LISTING 9.10 ReadFileUsingJFileChooser.java

```
1 import java.util.Scanner;
2 import javax.swing.JFileChooser;
3
4 public class ReadFileUsingJFileChooser {
5   public static void main(String[] args) throws Exception {
6     JFileChooser fileChooser = new JFileChooser();                 create a JFileChooser
7     if (fileChooser.showOpenDialog(null)                           display file chooser
```

check status

getSelectedFile

```
8         == JFileChooser.APPROVE_OPTION) {
9       // Get the selected file
10      java.io.File file = fileChooser.getSelectedFile();
11
12      // Create a Scanner for the file
13      Scanner input = new Scanner(file);
14
15      // Read text from the file
16      while (input.hasNext()) {
17        System.out.println(input.nextLine());
18      }
19
20      // Close the file
21      input.close();
22    }
23    else {
24      System.out.println("No file selected");
25    }
26  }
27 }
```

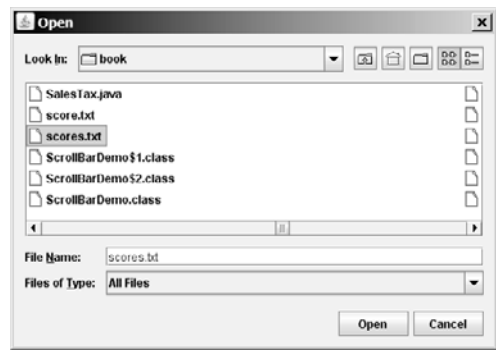


FIGURE 9.19 `JFileChooser` can be used to display a file dialog for opening a file.

showOpenDialog

APPROVE_OPTION

getSelectedFile

The program creates a `JFileChooser` in line 6. The `showOpenDialog(null)` method displays a dialog box, as shown in Figure 9.19. The method returns an `int` value, either `APPROVE_OPTION` or `CANCEL_OPTION`, which indicates whether the *Open* button or the *Cancel* button was clicked.

The `getSelectedFile()` method (line 10) returns the selected file from the file dialog box. Line 13 creates a scanner for the file. The program continuously reads the lines from the file and displays them to the console (lines 16–18).

CHAPTER SUMMARY

1. Strings are objects encapsulated in the `String` class. A string can be constructed using one of the 11 constructors or using a string literal shorthand initializer.

2. A `String` object is immutable; its contents cannot be changed. To improve efficiency and save memory, the JVM stores two literal strings that have the same character sequence in a unique object. This unique object is called an interned string object.

3. You can get the length of a string by invoking its `length()` method, retrieve a character at the specified `index` in the string using the `charAt(index)` method, and use the `indexOf` and `lastIndexOf` methods to find a character or a substring in a string.

4. You can use the `concat` method to concatenate two strings, or the plus (+) sign to concatenate two or more strings.

5. You can use the `substring` method to obtain a substring from the string.

6. You can use the `equals` and `compareTo` methods to compare strings. The `equals` method returns `true` if two strings are equal, and `false` if they are not equal. The `compareTo` method returns `0`, a positive integer, or a negative integer, depending on whether one string is equal to, greater than, or less than the other string.

7. The `Character` class is a wrapper class for a single character. The `Character` class provides useful static methods to determine whether a character is a letter (`isLetter(char)`), a digit (`isDigit(char)`), uppercase (`isUpperCase(char)`), or lowercase (`isLowerCase(char)`).

8. The `StringBuilder`/`StringBuffer` class can be used to replace the `String` class. The `String` object is immutable, but you can add, insert, or append new contents into a `StringBuilder`/`StringBuffer` object. Use `String` if the string contents do not require any change, and use `StringBuilder`/`StringBuffer` if they change.

9. You can pass strings to the `main` method from the command line. Strings passed to the `main` program are stored in `args`, which is an array of strings. The first string is represented by `args[0]`, and `args.length` is the number of strings passed.

10. The `File` class is used to obtain file properties and manipulate files. It does not contain the methods for creating a file or for reading/writing data from/to a file.

11. You can use `Scanner` to read string and primitive data values from a text file and use `PrintWriter` to create a file and write data to a text file.

12. The `JFileChooser` class can be used to display files graphically.

Review Questions

Section 9.2

9.1 Suppose that `s1`, `s2`, `s3`, and `s4` are four strings, given as follows:

```
String s1 = "Welcome to Java";
String s2 = s1;
String s3 = new String("Welcome to Java");
String s4 = "Welcome to Java";
```

What are the results of the following expressions?

```
(1) s1 == s2
(2) s2 == s3
(3) s1.equals(s2)
(4) s2.equals(s3)
(5) s1.compareTo(s2)
(6) s2.compareTo(s3)
(7) s1 == s4
(8) s1.charAt(0)
(9) s1.indexOf('j')
(10) s1.indexOf("to")
(11) s1.lastIndexOf('a')
(12) s1.lastIndexOf("o", 15)
(13) s1.length()
(14) s1.substring(5)
(15) s1.substring(5, 11)
(16) s1.startsWith("Wel")
```

```
(17) s1.endsWith("Java")      (21) s1.replace('o', 'T')
(18) s1.toLowerCase()         (22) s1.replaceAll("o", "T")
(19) s1.toUpperCase()         (23) s1.replaceFirst("o", "T")
(20) "  Welcome ".trim()      (24) s1.toCharArray()
```

To create a string "Welcome to Java", you may use a statement like this:

```
String s = "Welcome to Java";
```

or

```
String s = new String("Welcome to Java);
```

Which one is better? Why?

9.2 Suppose that `s1` and `s2` are two strings. Which of the following statements or expressions are incorrect?

```
String s = new String("new string");
String s3 = s1 + s2;
String s3 = s1 - s2;
s1 == s2;
s1 >= s2;
s1.compareTo(s2);
int i = s1.length();
char c = s1(0);
char c = s1.charAt(s1.length());
```

9.3 What is the printout of the following code?

```
String s1 = "Welcome to Java";
String s2 = s1.replace("o", "abc");
System.out.println(s1);
System.out.println(s2);
```

9.4 Let `s1` be `" Welcome "` and `s2` be `" welcome "`. Write the code for the following statements:

- Check whether `s1` is equal to `s2` and assign the result to a Boolean variable `isEqual`.
- Check whether `s1` is equal to `s2`, ignoring case, and assign the result to a Boolean variable `isEqual`.
- Compare `s1` with `s2` and assign the result to an `int` variable `x`.
- Compare `s1` with `s2`, ignoring case, and assign the result to an `int` variable `x`.
- Check whether `s1` has prefix `"AAA"` and assign the result to a Boolean variable `b`.
- Check whether `s1` has suffix `"AAA"` and assign the result to a Boolean variable `b`.
- Assign the length of `s1` to an `int` variable `x`.
- Assign the first character of `s1` to a `char` variable `x`.
- Create a new string `s3` that combines `s1` with `s2`.
- Create a substring of `s1` starting from index `1`.
- Create a substring of `s1` from index `1` to index `4`.
- Create a new string `s3` that converts `s1` to lowercase.
- Create a new string `s3` that converts `s1` to uppercase.
- Create a new string `s3` that trims blank spaces on both ends of `s1`.
- Replace all occurrences of character `e` with `E` in `s1` and assign the new string to `s3`.
- Split `"Welcome to Java and HTML"` into an array `tokens` delimited by a space.
- Assign the index of the first occurrence of character `e` in `s1` to an `int` variable `x`.
- Assign the index of the last occurrence of string `abc` in `s1` to an `int` variable `x`.

9.5 Does any method in the `String` class change the contents of the string?

9.6 Suppose string `s` is created using `new String()`; what is `s.length()`?

9.7 How do you convert a `char`, an array of characters, or a number to a string?

9.8 Why does the following code cause a `NullPointerException`?

```
1 public class Test {
2   private String text;
3
4   public Test(String s) {
5     String text  = s;
6   }
7
8   public static void main(String[] args) {
9     Test test = new Test("ABC");
10    System.out.println(test.text.toLowerCase());
11  }
12 }
```

9.9 What is wrong in the following program?

```
1 public class Test  {
2   String text;
3
4   public void Test(String s) {
5     this.text  = s;
6   }
7
8   public static void main(String[] args) {
9     Test test = new Test("ABC");
10    System.out.println(test);
11  }
12 }
```

Section 9.3

9.10 How do you determine whether a character is in lowercase or uppercase?

9.11 How do you determine whether a character is alphanumeric?

Section 9.4

9.12 What is the difference between `StringBuilder` and `StringBuffer`?

9.13 How do you create a string builder for a string? How do you get the string from a string builder?

9.14 Write three statements to reverse a string `s` using the `reverse` method in the `StringBuilder` class.

9.15 Write a statement to delete a substring from a string `s` of `20` characters, starting at index `4` and ending with index `10`. Use the `delete` method in the `StringBuilder` class.

9.16 What is the internal structure of a string and a string builder?

9.17 Suppose that `s1` and `s2` are given as follows:

```
StringBuilder s1 = new StringBuilder("Java");
StringBuilder s2 = new StringBuilder("HTML");
```

Show the value of `s1` after each of the following statements. Assume that the statements are independent.

```
(1) s1.append(" is fun");     (7) s1.deleteCharAt(3);
(2) s1.append(s2);            (8) s1.delete(1, 3);
(3) s1.insert(2, "is fun");   (9) s1.reverse();
(4) s1.insert(1, s2);         (10) s1.replace(1, 3, "Computer");
(5) s1.charAt(2);             (11) s1.substring(1, 3);
(6) s1.length();              (12) s1.substring(2);
```

9.18 Show the output of the following program:

```
public class Test {
  public static void main(String[] args) {
    String s = "Java";
    StringBuilder builder = new StringBuilder(s);
    change(s, builder);

    System.out.println(s);
    System.out.println(builder);
  }

  private static void change(String s, StringBuilder builder) {
    s = s + " and HTML";
    builder.append(" and HTML");
  }
}
```

Section 9.5

9.19 This book declares the `main` method as

```
public static void main(String[] args)
```

Can it be replaced by one of the following lines?

```
public static void main(String args[])

public static void main(String[] x)

public static void main(String x[])

static void main(String x[])
```

9.20 Show the output of the following program when invoked using

1. **java Test I have a dream**
2. **java Test "1 2 3"**
3. **java Test**
4. **java Test "*"**
5. **java Test ***

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Number of strings is " + args.length);
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

Section 9.6

9.21 What is wrong about creating a `File` object using the following statement?

```
new File("c:\book\test.dat");
```

9.22 How do you check whether a file already exists? How do you delete a file? How do you rename a file? Can you find the file size (the number of bytes) using the `File` class?

9.23 Can you use the `File` class for I/O? Does creating a `File` object create a file on the disk?

Section 9.7

9.24 How do you create a `PrintWriter` to write data to a file? What is the reason to declare `throws Exception` in the main method in Listing 9.7, WriteData.java? What would happen if the `close()` method were not invoked in Listing 9.7?

9.25 Show the contents of the file temp.txt after the following program is executed.

```
public class Test {
  public static void main(String[] args) throws Exception {
    java.io.PrintWriter output = new
      java.io.PrintWriter("temp.txt");
    output.printf("amount is %f %e\r\n", 32.32, 32.32);
    output.printf("amount is %5.4f %5.4e\r\n", 32.32, 32.32);
    output.printf("%6b\r\n", (1 > 2));
    output.printf("%6s\r\n", "Java");
    output.close();
  }
}
```

9.26 How do you create a `Scanner` to read data from a file? What is the reason to define `throws Exception` in the main method in Listing 9.8, ReadData.java? What would happen if the `close()` method were not invoked in Listing 9.8?

9.27 What will happen if you attempt to create a `Scanner` for a nonexistent file? What will happen if you attempt to create a `PrintWriter` for an existing file?

9.28 Is the line separator the same on all platforms? What is the line separator on Windows?

9.29 Suppose you enter `45 57.8 789`, then press the *Enter* key. Show the contents of the variables after the following code is executed.

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
double doubleValue = input.nextDouble();
String line = input.nextLine();
```

9.30 Suppose you enter `45`, the *Enter* key, `57.8`, the *Enter* key, `789`, the *Enter* key. Show the contents of the variables after the following code is executed.

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
double doubleValue = input.nextDouble();
String line = input.nextLine();
```

Programming Exercises

Sections 9.2–9.3

9.1* (*Checking SSN*) Write a program that prompts the user to enter a social security number in the format DDD-DD-DDDD, where D is a digit. The program displays `"Valid SSN"` for a correct social security number and `"Invalid SSN"` otherwise.

9.2** (*Checking substrings*) You can check whether a string is a substring of another string by using the `indexOf` method in the `String` class. Write your own method for this function. Write a program that prompts the user to enter two strings, and check whether the first string is a substring of the second.

9.3** (*Checking password*) Some Websites impose certain rules for passwords. Write a method that checks whether a string is a valid password. Suppose the password rule is as follows:

- A password must have at least eight characters.
- A password consists of only letters and digits.
- A password must contain at least two digits.

Write a program that prompts the user to enter a password and displays `"Valid Password"` if the rule is followed or `"Invalid Password"` otherwise.

9.4 (*Occurrences of a specified character*) Write a method that finds the number of occurrences of a specified character in the string using the following header:

```
public static int count(String str, char a)
```

For example, `count("Welcome", 'e')` returns `2`. Write a test program that prompts the user to enter a string followed by a character and displays the number of occurrences of the character in the string.

9.5** (*Occurrences of each digit in a string*) Write a method that counts the occurrences of each digit in a string using the following header:

```
public static int[] count(String s)
```

The method counts how many times a digit appears in the string. The return value is an array of ten elements, each of which holds the count for a digit. For example, after executing `int[] counts = count("12203AB3")`, `counts[0]` is `1`, `counts[1]` is `1`, `counts[2]` is `2`, `counts[3]` is `2`.

Write a test program that prompts the user to enter a string and displays the number of occurrences of each digit in the string.

9.6* (*Counting the letters in a string*) Write a method that counts the number of letters in a string using the following header:

```
public static int countLetters(String s)
```

Write a test program that prompts the user to enter a string and displays the number of letters in the string.

9.7* (*Phone keypads*) The international standard letter/number mapping found on the telephone is shown below:

1	2	3
	ABC	DEF
4	5	6
GHI	JKL	MNO
7	8	9
PQRS	TUV	WXYZ
	0	

Write a method that returns a number, given an uppercase letter, as follows:

```
public static int getNumber(char uppercaseLetter)
```

Write a test program that prompts the user to enter a phone number as a string. The input number may contain letters. The program translates a letter (upper- or lowercase) to a digit and leaves all other characters intact. Here is a sample run of the program:

```
Enter a string: 1-800-Flowers
1-800-3569377
```



```
Enter a string: 1800flowers
18003569377
```



9.8* (*Binary to decimal*) Write a method that parses a binary number as a string into a decimal integer. The method header is as follows:

```
public static int binaryToDecimal(String binaryString)
```

For example, binary string 10001 is 17 ($1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2 + 1 = 17$). So, `binaryToDecimal("10001")` returns `17`. Note that `Integer.parseInt("10001", 2)` parses a binary string to a decimal value. Do not use this method in this exercise.

Write a test program that prompts the user to enter a binary string and displays the corresponding decimal integer value.

Section 9.4

9.9** (*Binary to hex*) Write a method that parses a binary number into a hex number. The method header is as follows:

```
public static String binaryToHex(String binaryValue)
```

Write a test program that prompts the user to enter a binary number and displays the corresponding hexadecimal value.

9.10** (*Decimal to binary*) Write a method that parses a decimal number into a binary number as a string. The method header is as follows:



Video Note
Number conversion

```
public static String decimalToBinary(int value)
```

Write a test program that prompts the user to enter a decimal integer value and displays the corresponding binary value.

9.11** (*Sorting characters in a string*) Write a method that returns a sorted string using the following header:

```
public static String sort(String s)
```

For example, `sort("acb")` returns `abc`.

Write a test program that prompts the user to enter a string and displays the sorted string.

9.12** (*Anagrams*) Write a method that checks whether two words are anagrams. Two words are anagrams if they contain the same letters in any order. For example, `"silent"` and `"listen"` are anagrams. The header of the method is as follows:

```
public static boolean isAnagram(String s1, String s2)
```

Write a test program that prompts the user to enter two strings and, if they are anagrams, displays `"anagram"`, otherwise displays `"not anagram"`.

Section 9.5

9.13* (*Passing a string to check palindromes*) Rewrite Listing 9.1 by passing the string as a command-line argument.

9.14* (*Summing integers*) Write two programs. The first program passes an unspecified number of integers as separate strings to the `main` method and displays their total. The second program passes an unspecified number of integers delimited by one space in a string to the `main` method and displays their total. Name the two programs `Exercise9_14a` and `Exercise9_14b`, as shown in Figure 9.20.

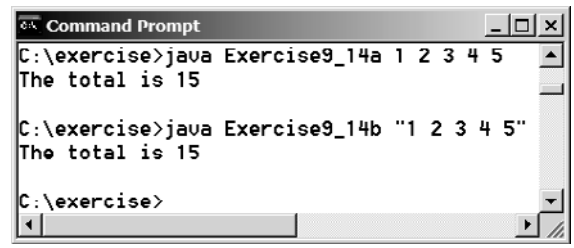

Command Prompt
C:\exercise>java Exercise9_14a 1 2 3 4 5
The total is 15
C:\exercise>java Exercise9_14b "1 2 3 4 5"
The total is 15
C:\exercise>

FIGURE 9.20 The program adds all the numbers passed from the command line.

9.15* (*Finding the number of uppercase letters in a string*) Write a program that passes a string to the `main` method and displays the number of uppercase letters in a string.

Sections 9.7–9.8

9.16** (*Reformatting Java source code*) Write a program that converts the Java source code from the next-line brace style to the end-of-line brace style. For example, the Java source in (a) below uses the next-line brace style. Your program converts it to the end-of-line brace style in (b).

```
public class Test
{
  public static void main(String[] args)
  {
    // Some statements
  }
}
```

(a) Next-line brace style

```
public class Test {
  public static void main(String[] args) {
    // Some statements
  }
}
```

(b) End-of-line brace style

Your program can be invoked from the command line with the Java source-code file as the argument. It converts the Java source code to a new format. For example, the following command converts the Java source-code file **Test.java** to the end-of-line brace style.

```
java Exercise9_16 Test.java
```

9.17* (*Counting characters, words, and lines in a file*) Write a program that will count the number of characters (excluding control characters `'\r'` and `'\n'`), words, and lines, in a file. Words are separated by spaces, tabs, carriage return, or line-feed

characters. The file name should be passed as a command-line argument, as shown in Figure 9.21.

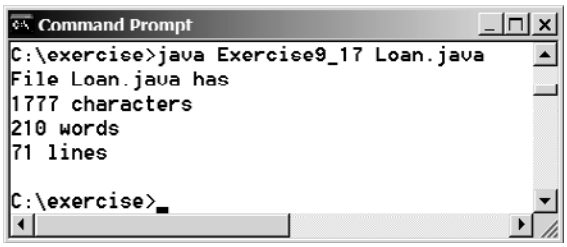


FIGURE 9.21 The program displays the number of characters, words, and lines in the given file.

9.18* (*Processing scores in a text file*) Suppose that a text file **Exercise9_18.txt** contains an unspecified number of scores. Write a program that reads the scores from the file and displays their total and average. Scores are separated by blanks.

9.19* (*Writing/Reading data*) Write a program to create a file named **Exercise9_19.txt** if it does not exist. Write **100** integers created randomly into the file using text I/O. Integers are separated by spaces in the file. Read the data back from the file and display the sorted data.

9.20** (*Replacing text*) Listing 9.9, ReplaceText.java, gives a program that replaces text in a source file and saves the change into a new file. Revise the program to save the change into the original file. For example, invoking

```
java Exercise9_20 file oldString newString
```

replaces `oldString` in the source file with `newString`.

9.21** (*Removing text*) Write a program that removes all the occurrences of a specified string from a text file. For example, invoking

```
java Exercise9_21 John filename
```

removes string `John` from the specified file.

Comprehensive

9.22** (*Guessing the capitals*) Write a program that repeatedly prompts the user to enter a capital for a state, as shown in Figure 9.22(a). Upon receiving the user input, the program reports whether the answer is correct, as shown in Figure 9.22(b). Assume that **50** states and their capitals are stored in a two-dimensional array, as shown in Figure 9.23. The program prompts the user to answer all ten states' capitals and displays the total correct count.



(a) (b)

FIGURE 9.22 The program prompts the user to enter the capital in (a) and reports the correctness of the answer.

Alabama	Montgomery
Alaska	Juneau
Arizona	Phoenix
...	...
...	...

FIGURE 9.23 A two-dimensional array stores states and their capitals.

9.23** (*Implementing the `String` class*) The `String` class is provided in the Java library. Provide your own implementation for the following methods (name the new class `MyString1`):

```
public MyString1(char[] chars);
public char charAt(int index);
public int length();
public MyString1 substring(int begin, int end);
public MyString1 toLowerCase();
public boolean equals(MyString1 s);
public static MyString1 valueOf(int i);
```

9.24** (*Implementing the `String` class*) The `String` class is provided in the Java library. Provide your own implementation for the following methods (name the new class `MyString2`):

```
public MyString2(String s);
public int compare(String s);
public MyString2 substring(int begin);
public MyString2 toUpperCase();
public char[] toChars();
public static MyString2 valueOf(boolean b);
```

9.25 (*Implementing the `Character` class*) The `Character` class is provided in the Java library. Provide your own implementation for this class. Name the new class `MyCharacter`.

9.26** (*Implementing the `StringBuilder` class*) The `StringBuilder` class is provided in the Java library. Provide your own implementation for the following methods (name the new class `MyStringBuilder1`):

```
public MyStringBuilder1(String s);
public MyStringBuilder1 append(MyStringBuilder1 s);
public MyStringBuilder1 append(int i);
public int length();
public char charAt(int index);
public MyStringBuilder1 toLowerCase();
public MyStringBuilder1 substring(int begin, int end);
public String toString();
```

9.27** (*Implementing the `StringBuilder` class*) The `StringBuilder` class is provided in the Java library. Provide your own implementation for the following methods (name the new class `MyStringBuilder2`):

```
public MyStringBuilder2();
public MyStringBuilder2(char[] chars);
public MyStringBuilder2(String s);
public MyStringBuilder2 insert(int offset, MyStringBuilder2 s);
```

```
public MyStringBuilder2 reverse();
public MyStringBuilder2 substring(int begin);
public MyStringBuilder2 toUpperCase();
```

9.28* (*Common prefix*) Write a method that returns the common prefix of two strings. For example, the common prefix of `"distance"` and `"disinfection"` is `"dis"`. The header of the method is as follows:

```
public static String prefix(String s1, String s2)
```

If the two strings have no common prefix, the method returns an empty string.

Write a `main` method that prompts the user to enter two strings and display their common prefix.

9.29** (*New string `split` method*) The `split` method in the `String` class returns an array of strings consisting of the substrings split by the delimiters. However, the delimiters are not returned. Implement the following new method that returns an array of strings consisting of the substrings split by the matches, including the matches.

```
public static String[] split(String s, String regex)
```

For example, `split("ab#12#453", "#")` returns `ab`, `#`, `12`, `#`, `453` in an array of `String`, and `split("a?b?gf#e", "[?#]")` returns `a`, `b`, `?`, `b`, `gf`, `#`, and `e` in an array of `String`.

9.30** (*Financial: credit card number validation*) Rewrite Exercise 5.31 using a string input for credit card number. Redesign the program using the following method:

```
/** Return true if the card number is valid */
public static boolean isValid(String cardNumber)

/** Get the result from Step 2 */
public static int sumOfDoubleEvenPlace(String cardNumber)

/** Return this number if it is a single digit; otherwise,
  * return the sum of the two digits */
public static int getDigit(int number)

/** Return sum of odd place digits in number */
public static int sumOfOddPlace(String cardNumber)
```

9.31*** (*Game: hangman*) Write a hangman game that randomly generates a word and prompts the user to guess one letter at a time, as shown in the sample run. Each letter in the word is displayed as an asterisk. When the user makes a correct guess, the actual letter is then displayed. When the user finishes a word, display the number of misses and ask the user whether to continue for another word. Declare an array to store words, as follows:

```
// Use any words you wish
String[] words = {"write", "that", ...};
```



```
(Guess) Enter a letter in word ******* > p
(Guess) Enter a letter in word p****** > r
(Guess) Enter a letter in word pr**r** > p
     p  is already in the word
(Guess) Enter a letter in word pr**r** > o
(Guess) Enter a letter in word pro*r** > g
(Guess) Enter a letter in word progr** > n
     n  is not in the word
(Guess) Enter a letter in word progr** > m
(Guess) Enter a letter in word progr*m > a
The word is program. You missed 1 time

Do you want to guess for another word? Enter y or n>
```

9.32** (*Checking ISBN*) Use string operations to simplify Exercise 3.9. Enter the first 9 digits of an ISBN number as a string.

9.33** (*Game: hangman*) Rewrite Exercise 9.31. The program reads the words stored in a text file named **Exercise9_33.txt**. Words are delimited by spaces.

9.34** (*Replacing text*) Revise Exercise9_20 to replace a string in a file with a new string for all files in the specified directory using the following command:

java Exercise9_34 dir oldString newString

9.35* (*Bioinformatics: finding genes*) Biologists use a sequence of letters **A**, **C**, **T**, and **G** to model a genome. A gene is a substring of a genome that starts after a triplet **ATG** and ends before a triplet **TAG**, **TAA**, or **TGA**. Furthermore, the length of a gene string is a multiple of 3 and the gene does not contain any of the triplets **ATG**, **TAG**, **TAA**, and **TGA**. Write a program that prompts the user to enter a genome and displays all genes in the genome. If no gene is found in the input sequence, displays no gene. Here are the sample runs:



```
Enter a genome string: TTATGTTTTAAGGATGGGGCGTTAGTT Enter
TTT
GGGCGT
```



```
Enter a genome string: TGTGTGTATAT Enter
no gene is found
```

CHAPTER 10

THINKING IN OBJECTS

Objectives

- To create immutable objects from immutable classes to protect the contents of objects (§10.2).
- To determine the scope of variables in the context of a class (§10.3).
- To use the keyword `this` to refer to the calling object itself (§10.4).
- To apply class abstraction to develop software (§10.5).
- To explore the differences between the procedural paradigm and object-oriented paradigm (§10.6).
- To develop classes for modeling composition relationships (§10.7).
- To design programs using the object-oriented paradigm (§§10.8–10.10).
- To design classes that follow the class-design guidelines (§10.11).



10.1 Introduction

The preceding two chapters introduced objects and classes. You learned how to define classes, create objects, and use objects from several classes in the Java API (e.g., `Date`, `Random`, `String`, `StringBuilder`, `File`, `Scanner`, `PrintWriter`). This book's approach is to teach problem solving and fundamental programming techniques before object-oriented programming. This chapter will show how procedural and object-oriented programming differ. You will see the benefits of object-oriented programming and learn to use it effectively.

Our focus here is on class design. We will use several examples to illustrate the advantages of the object-oriented approach. The examples involve designing new classes and using them in applications. We first introduce some language features supporting these examples.

10.2 Immutable Objects and Classes

Normally, you create an object and allow its contents to be changed later. Occasionally it is desirable to create an object whose contents cannot be changed, once the object is created. We call such an object an *immutable object* and its class an *immutable class*. The `String` class, for example, is immutable. If you deleted the `set` method in the `Circle` class in Listing 8.9, the class would be immutable, because radius is private and cannot be changed without a `set` method.

immutable object
immutable class

If a class is immutable, then all its data fields must be private and it cannot contain public `set` methods for any data fields. A class with all private data fields and no mutators is not necessarily immutable. For example, the following `Student` class has all private data fields and no `set` methods, but it is not an immutable class.

`Student` class

```
1  public class Student {
2    private int id;
3    private String name;
4    private java.util.Date dateCreated;
5
6    public Student(int ssn, String newName) {
7      id = ssn;
8      name = newName;
9      dateCreated = new java.util.Date();
10   }
11
12   public int getId() {
13     return id;
14   }
15
16   public String getName() {
17     return name;
18   }
19
20   public java.util.Date getDateCreated() {
21     return dateCreated;
22   }
23 }
```

As shown in the code below, the data field `dateCreated` is returned using the `getDateCreated()` method. This is a reference to a `Date` object. Through this reference, the content for `dateCreated` can be changed.

```
public class Test {
  public static void main(String[] args) {
    Student student = new Student(111223333, "John");
    java.util.Date dateCreated = student.getDateCreated();
```

```
    dateCreated.setTime(200000); // Now dateCreated field is changed!
  }
}
```

For a class to be immutable, it must meet the following requirements:

- all data fields private;
- no mutator methods;
- no accessor method that returns a reference to a data field that is mutable.

10.3 The Scope of Variables

Chapter 5, "Methods," discussed local variables and their scope rules. Local variables are declared and used inside a method locally. This section discusses the scope rules of all the variables in the context of a class.

Instance and static variables in a class are referred to as the *class's variables* or *data fields*. A variable defined inside a method is referred to as a local variable. The scope of a class's variables is the entire class, regardless of where the variables are declared. A class's variables and methods can appear in any order in the class, as shown in Figure 10.1(a). The exception is when a data field is initialized based on a reference to another data field. In such cases, the other data field must be declared first, as shown in Figure 10.1(b). For consistency, this book declares data fields at the beginning of the class.

```
public class Circle {
  public double findArea() {
    return radius * radius * Math.PI;
  }

  private double radius = 1;
}
```

(a) variable `radius` and method `findArea()` can be declared in any order

```
public class Foo {
  private int i;
  private int j = i + 1;
}
```

(b) `i` has to be declared before `j` because `j`'s initial value is dependent on `i`.

FIGURE 10.1 Members of a class can be declared in any order, with one exception.

You can declare a class's variable only once, but you can declare the same variable name in a method many times in different nonnesting blocks.

If a local variable has the same name as a class's variable, the local variable takes precedence and the class's variable with the same name is *hidden*. For example, in the following program, `x` is defined as an instance variable and as a local variable in the method.

```
public class Foo {
  private int x = 0; // Instance variable
  private int y = 0;

  public Foo() {
  }

  public void p() {
    int x = 1; // Local variable
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  }
}
```

What is the printout for `f.p()`, where `f` is an instance of `Foo`? The printout for `f.p()` is `1` for `x` and `0` for `y`. Here is why:

- `x` is declared as a data field with the initial value of `0` in the class, but is also declared in the method `p()` with an initial value of `1`. The latter `x` is referenced in the `System.out.println` statement.
- `y` is declared outside the method `p()`, but is accessible inside it.



Tip

To avoid confusion and mistakes, do not use the names of instance or static variables as local variable names, except for method parameters.

10.4 The `this` Reference

hidden data fields

The `this` keyword is the name of a reference that refers to a calling object itself. One of its common uses is to reference a class's *hidden data fields*. For example, a data-field name is often used as the parameter name in a `set` method for the data field. In this case, the data field is hidden in the `set` method. You need to reference the hidden data-field name in the method in order to set a new value to it. A hidden static variable can be accessed simply by using the `ClassName.StaticVariable` reference. A hidden instance variable can be accessed by using the keyword `this`, as shown in Figure 10.2(a).

```
public class Foo {
  int i = 5;
  static double k = 0;

  void setI(int i) {
    this.i = i;
  }

  static void setK(double k) {
    Foo.k = k;
  }
}
```

(a)

```
Suppose that f1 and f2 are two objects of Foo.

Invoking f1.setI(10) is to execute
   this.i = 10, where this refers f1

Invoking f2.setI(45) is to execute
   this.i = 45, where this refers f2
```

(b)

FIGURE 10.2 The keyword `this` refers to the calling object that invokes the method.

The `this` keyword gives us a way to refer to the object that invokes an instance method within the code of the instance method. The line `this.i = i` means "assign the value of parameter `i` to the data field `i` of the calling object." The keyword `this` refers to the object that invokes the instance method `setI`, as shown in Figure 10.2(b). The line `Foo.k = k` means that the value in parameter `k` is assigned to the static data field `k` of the class, which is shared by all the objects of the class.

call another constructor

Another common use of the `this` keyword is to enable a constructor to invoke another constructor of the same class. For example, you can rewrite the `Circle` class as follows:

```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  public double getArea() {
    return this.radius * this.radius * Math.PI;
  }
}
```

`this` must be explicitly used to reference the data field radius of the object being constructed

`this` is used to invoke another constructor

Every instance variable belongs to an instance represented by `this`, which is normally omitted

The line `this(1.0)` in the second constructor invokes the first constructor with a `double` value argument.



Tip

If a class has multiple constructors, it is better to implement them using `this(arg-list)` as much as possible. In general, a constructor with no or fewer arguments can invoke the constructor with more arguments using `this(arg-list)`. This often simplifies coding and makes the class easier to read and to maintain.



Note

Java requires that the `this(arg-list)` statement appear first in the constructor before any other statements.

10.5 Class Abstraction and Encapsulation

In Chapter 5, "Methods," you learned about method abstraction and used it in program development. Java provides many levels of abstraction. *Class abstraction* is the separation of class implementation from the use of a class. The creator of a class describes it and lets the user know how it can be used. The collection of methods and fields that are accessible from outside the class, together with the description of how these members are expected to behave, serves as the *class's contract*. As shown in Figure 10.3, the user of the class does not need to know how the class is implemented. The details of implementation are encapsulated and hidden from the user. This is known as *class encapsulation*. For example, you can create a `Circle` object and find the area of the circle without knowing how the area is computed.

class abstraction

class encapsulation

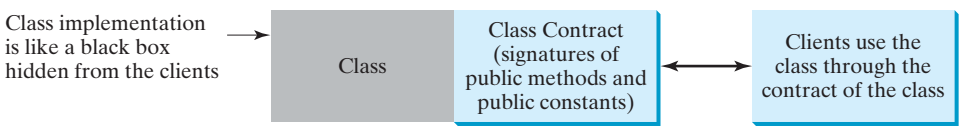

Class implementation is like a black box hidden from the clients
Class
Class Contract (signatures of public methods and public constants)
Clients use the class through the contract of the class

FIGURE 10.3 Class abstraction separates class implementation from the use of the class.

Class abstraction and encapsulation are two sides of the same coin. Many real-life examples illustrate the concept of class abstraction. Consider, for instance, building a computer system. Your personal computer has many components—a CPU, memory, disk, motherboard, fan, and so on. Each component can be viewed as an object that has properties and methods. To get the components to work together, you need know only how each component is used and how it interacts with the others. You don't need to know how the components work internally. The internal implementation is encapsulated and hidden from you. You can build a computer without knowing how a component is implemented.

The computer-system analogy precisely mirrors the object-oriented approach. Each component can be viewed as an object of the class for the component. For example, you might have a class that models all kinds of fans for use in a computer, with properties such as fan size and speed and methods such as start and stop. A specific fan is an instance of this class with specific property values.

As another example, consider getting a loan. A specific loan can be viewed as an object of a `Loan` class. Interest rate, loan amount, and loan period are its data properties, and computing monthly payment and total payment are its methods. When you buy a car, a loan object is created by instantiating the class with your loan interest rate, loan amount, and loan period. You can then use the methods to find the monthly payment and total payment of your loan. As a user of the `Loan` class, you don't need to know how these methods are implemented.

Listing 2.8, ComputeLoan.java, presented a program for computing loan payments. The program cannot be reused in other programs. One way to fix this problem is to define static methods for computing monthly payment and total payment. However, this solution has limitations. Suppose you wish to associate a date with the loan. The ideal way is to create an



Video Note
The **Loan** class

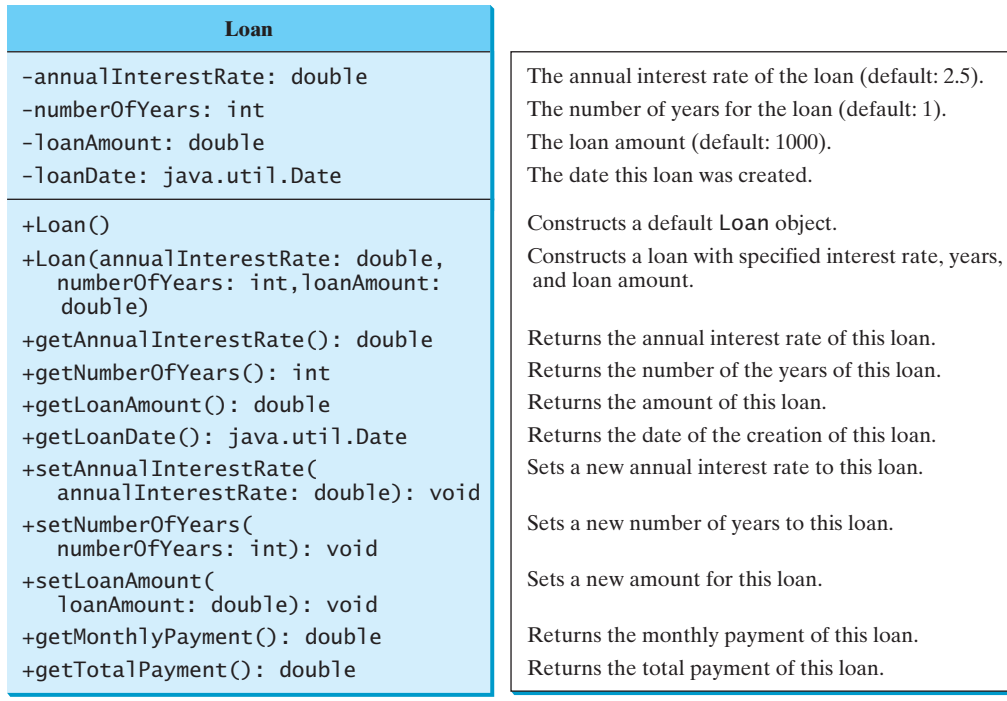

Loan
-annualInterestRate: double
-numberOfYears: int
-loanAmount: double
-loanDate: java.util.Date
+Loan()
+Loan(annualInterestRate: double, numberOfYears: int,loanAmount: double)
+getAnnualInterestRate(): double
+getNumberOfYears(): int
+getLoanAmount(): double
+getLoanDate(): java.util.Date
+setAnnualInterestRate(annualInterestRate: double): void
+setNumberOfYears(numberOfYears: int): void
+setLoanAmount(loanAmount: double): void
+getMonthlyPayment(): double
+getTotalPayment(): double
The annual interest rate of the loan (default: 2.5).
The number of years for the loan (default: 1).
The loan amount (default: 1000).
The date this loan was created.
Constructs a default Loan object.
Constructs a loan with specified interest rate, years, and loan amount.
Returns the annual interest rate of this loan.
Returns the number of the years of this loan.
Returns the amount of this loan.
Returns the date of the creation of this loan.
Sets a new annual interest rate to this loan.
Sets a new number of years to this loan.
Sets a new amount for this loan.
Returns the monthly payment of this loan.
Returns the total payment of this loan.

FIGURE 10.4 The **Loan** class models the properties and behaviors of loans.

object that ties the properties for loan information and date together. Figure 10.4 shows the UML class diagram for the **Loan** class.

The UML diagram in Figure 10.4 serves as the contract for the **Loan** class. Throughout this book, you will play the roles of both class user and class developer. The user can use the class without knowing how the class is implemented. Assume that the **Loan** class is available. We begin by writing a test program that uses the **Loan** class (Listing 10.1).

LISTING 10.1 TestLoanClass.java

```
1 import java.util.Scanner;
2
3 public class TestLoanClass {
4   /** Main method */
5   public static void main(String[] args) {
6     // Create a Scanner
7     Scanner input = new Scanner(System.in);
8
9     // Enter yearly interest rate
10    System.out.print(
11      "Enter yearly interest rate, for example, 8.25: ");
12    double annualInterestRate = input.nextDouble();
13
14    // Enter number of years
15    System.out.print("Enter number of years as an integer: ");
16    int numberOfYears = input.nextInt();
17
18    // Enter loan amount
19    System.out.print("Enter loan amount, for example, 120000.95: ");
```

```
20     double loanAmount =  input.nextDouble();
21
22     // Create a Loan object
23     Loan loan =
24       new Loan(annualInterestRate, numberOfYears, loanAmount);          create Loan object
25
26     // Display loan date, monthly payment, and total payment
27     System.out.printf("The loan was created on %s\n" +
28       "The monthly payment is %.2f\nThe total payment is %.2f\n",
29       loan.getLoanDate().toString(), loan.getMonthlyPayment(),          invoke instance method
30       loan.getTotalPayment());                                          invoke instance method
31   }
32 }
```

```
Enter yearly interest rate, for example, 8.25: 2.5 Enter
Enter number of years as an integer: 5 Enter
Enter loan amount, for example, 120000.95: 1000 Enter
The loan was created on Sat Jun 10 21:12:50 EDT 2006
The monthly payment is 17.74
The total payment is 1064.84
```



The `main` method reads interest rate, payment period (in years), and loan amount; creates a `Loan` object; and then obtains the monthly payment (line 29) and total payment (line 30) using the instance methods in the `Loan` class.

The `Loan` class can be implemented as in Listing 10.2.

LISTING 10.2 Loan.java

```
1  public class Loan {
2    private double annualInterestRate;
3    private int numberOfYears;
4    private double loanAmount;
5    private java.util.Date loanDate;
6
7    /** Default constructor */
8    public Loan() {                                                      no-arg constructor
9      this(2.5, 1, 1000);
10   }
11
12   /** Construct a loan with specified annual interest rate,
13       number of years, and loan amount
14     */
15   public Loan(double annualInterestRate, int numberOfYears,            constructor
16       double loanAmount) {
17     this.annualInterestRate = annualInterestRate;
18     this.numberOfYears = numberOfYears;
19     this.loanAmount = loanAmount;
20     loanDate = new java.util.Date();
21   }
22
23   /** Return annualInterestRate */
24   public double getAnnualInterestRate() {
25     return annualInterestRate;
26   }
```

```
27
28   /** Set a new annualInterestRate */
29   public void setAnnualInterestRate(double annualInterestRate) {
30     this.annualInterestRate = annualInterestRate;
31   }
32
33   /** Return numberOfYears */
34   public int getNumberOfYears() {
35     return numberOfYears;
36   }
37
38   /** Set a new numberOfYears */
39   public void setNumberOfYears(int numberOfYears) {
40     this.numberOfYears = numberOfYears;
41   }
42
43   /** Return loanAmount */
44   public double getLoanAmount() {
45     return loanAmount;
46   }
47
48   /** Set a newloanAmount */
49   public void setLoanAmount(double loanAmount) {
50     this.loanAmount = loanAmount;
51   }
52
53   /** Find monthly payment */
54   public double getMonthlyPayment() {
55     double monthlyInterestRate = annualInterestRate / 1200;
56     double monthlyPayment = loanAmount * monthlyInterestRate / (1 -
57       (Math.pow(1 / (1 + monthlyInterestRate), numberOfYears * 12)));
58     return monthlyPayment;
59   }
60
61   /** Find total payment */
62   public double getTotalPayment() {
63     double totalPayment = getMonthlyPayment() * numberOfYears * 12;
64     return totalPayment;
65   }
66
67   /** Return loan date */
68   public java.util.Date getLoanDate() {
69     return loanDate;
70   }
71 }
```

From a class developer's perspective, a class is designed for use by many different customers. In order to be useful in a wide range of applications, a class should provide a variety of ways for customization through constructors, properties, and methods.

The `Loan` class contains two constructors, four `get` methods, three `set` methods, and the methods for finding monthly payment and total payment. You can construct a `Loan` object by using the no-arg constructor or the one with three parameters: annual interest rate, number of years, and loan amount. When a loan object is created, its date is stored in the `loanDate` field. The `getLoanDate` method returns the date. The three `get` methods, `getAnnualInterest`, `getNumberOfYears`, and `getLoanAmount`, return annual interest rate, payment years, and loan amount, respectively. All the data properties and methods in this class are tied to a specific instance of the `Loan` class. Therefore, they are instance variables or methods.



Important Pedagogical Tip

The UML diagram for the **Loan** class is shown in Figure 10.4. Students should begin by writing a test program that uses the **Loan** class even though they don't know how the **Loan** class is implemented. This has three benefits:

- It demonstrates that developing a class and using a class are two separate tasks.
- It enables you to skip the complex implementation of certain classes without interrupting the sequence of the book.
- It is easier to learn how to implement a class if you are familiar with the class through using it.

For all the examples from now on, you may first create an object from the class and try to use its methods and then turn your attention to its implementation.

10.6 Object-Oriented Thinking

Chapters 1–7 introduced fundamental programming techniques for problem solving using loops, methods, and arrays. The study of these techniques lays a solid foundation for object-oriented programming. Classes provide more flexibility and modularity for building reusable software. This section improves the solution for a problem introduced in Chapter 3 using the object-oriented approach. From the improvements, you will gain insight on the differences between procedural and object-oriented programming and see the benefits of developing reusable code using objects and classes.

Listing 3.5, ComputeBMI.java, presented a program for computing body mass index. The code cannot be reused in other programs. To make it reusable, define a static method to compute body mass index as follows:

```
public static double getBMI(double weight, double height)
```

This method is useful for computing body mass index for a specified weight and height. However, it has limitations. Suppose you need to associate the weight and height with a person's name and birth date. You may declare separate variables to store these values. But these values are not tightly coupled. The ideal way to couple them is to create an object that contains them. Since these values are tied to individual objects, they should be stored in instance data fields. You can define a class named **BMI**, as shown in Figure 10.5.



Video Note
The BMI class

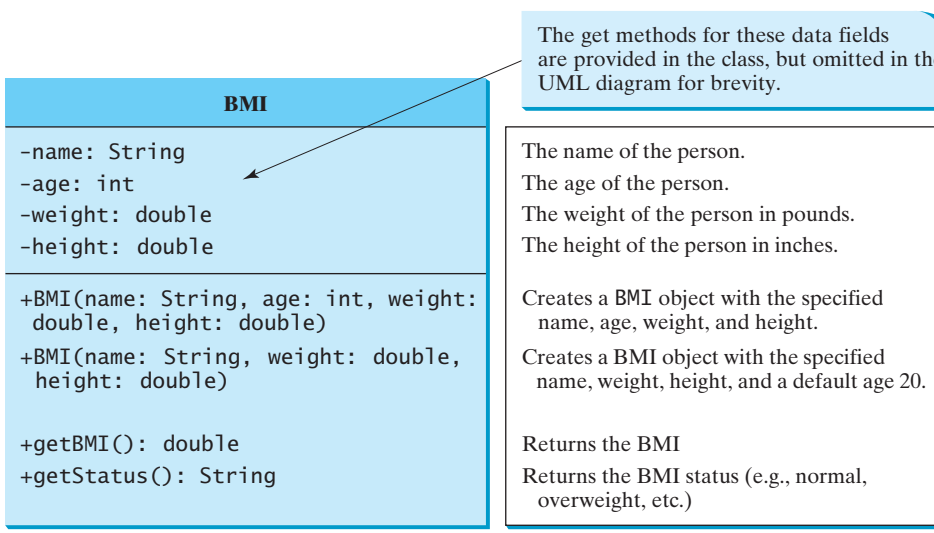

The get methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
BMI
-name: String
-age: int
-weight: double
-height: double
+BMI(name: String, age: int, weight: double, height: double)
+BMI(name: String, weight: double, height: double)
+getBMI(): double
+getStatus(): String
The name of the person.
The age of the person.
The weight of the person in pounds.
The height of the person in inches.
Creates a BMI object with the specified name, age, weight, and height.
Creates a BMI object with the specified name, weight, height, and a default age 20.
Returns the BMI
Returns the BMI status (e.g., normal, overweight, etc.)

FIGURE 10.5 The **BMI** class encapsulates BMI information.

Assume that the `BMI` class is available. Listing 10.3 gives a test program that uses this class.

LISTING 10.3 UseBMIClass.java

```
1  public class UseBMIClass {
2    public static void main(String[] args) {
3      BMI bmi1 = new BMI("John Doe", 18, 145, 70);
4      System.out.println("The BMI for " + bmi1.getName() + " is "
5        + bmi1.getBMI() + " " + bmi1.getStatus());
6
7      BMI bmi2 = new BMI("Peter King", 215, 70);
8      System.out.println("The BMI for " + bmi2.getName() + " is "
9        + bmi2.getBMI() + " " + bmi2.getStatus());
10   }
11 }
```

create an object
invoke instance method
create an object
invoke instance method



```
The BMI for John Doe is 20.81 normal weight
The BMI for Peter King is 30.85 seriously overweight
```

Line 3 creates an object `bmi1` for John Doe and line 7 creates an object `bmi2` for Peter King. You can use the instance methods `getName()`, `getBMI()`, and `getStatus()` to return the BMI information in a `BMI` object.

The `BMI` class can be implemented as in Listing 10.4.

LISTING 10.4 BMI.java

```
1  public class BMI {
2    private String name;
3    private int age;
4    private double weight; // in pounds
5    private double height; // in inches
6    public static final double KILOGRAMS_PER_POUND = 0.45359237;
7    public static final double METERS_PER_INCH = 0.0254;
8
9    public BMI(String name, int age, double weight, double height) {
10     this.name = name;
11     this.age = age;
12     this.weight = weight;
13     this.height = height;
14   }
15
16   public BMI(String name, double weight, double height) {
17     this(name, 20, weight, height);
18   }
19
20   public double getBMI() {
21     double bmi = weight * KILOGRAMS_PER_POUND /
22       ((height * METERS_PER_INCH) * (height * METERS_PER_INCH));
23     return Math.round(bmi * 100) / 100.0;
24   }
25
26   public String getStatus() {
27     double bmi = getBMI();
28     if (bmi < 16)
29       return "seriously underweight";
30     else if (bmi < 18)
```

constructor
constructor
getBMI
getStatus

```
31        return "underweight";
32      else if (bmi < 24)
33        return "normal weight";
34      else if (bmi < 29)
35        return "overweight";
36      else if (bmi < 35)
37        return "seriously overweight";
38      else
39        return "gravely overweight";
40    }
41
42    public String getName() {
43      return name;
44    }
45
46    public int getAge() {
47      return age;
48    }
49
50    public double getWeight() {
51      return weight;
52    }
53
54    public double getHeight() {
55      return height;
56    }
57  }
```

The mathematic formula for computing the BMI using weight and height is given in §3.10. The instance method `getBMI()` returns the BMI. Since the weight and height are instance data fields in the object, the `getBMI()` method can use these properties to compute the BMI for the object.

The instance method `getStatus()` returns a string that interprets the BMI. The interpretation is also given in §3.10.

Procedural vs. Object-Oriented Paradigms

This example demonstrates the advantages of the object-oriented paradigm over the procedural paradigm. The procedural paradigm focuses on designing methods. The object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

In procedural programming, data and operations on the data are separate, and this methodology requires sending data to methods. Object-oriented programming places data and the operations that pertain to them in an object. This approach solves many of the problems inherent in procedural programming. The object-oriented programming approach organizes programs in a way that mirrors the real world, in which all objects are associated with both attributes and activities. Using objects improves software reusability and makes programs easier to develop and easier to maintain. Programming in Java involves thinking in terms of objects; a Java program can be viewed as a collection of cooperating objects.

10.7 Object Composition

An object can contain another object. The relationship between the two is called *composition*. In Listing 10.4, you defined the `BMI` class to contain a `String` data field. The relationship between `BMI` and `String` is composition.

Composition is actually a special case of the aggregation relationship. Aggregation models *has-a* relationships and represents an ownership relationship between two objects. The owner

object is called an *aggregating object* and its class an *aggregating class*. The subject object is called an *aggregated object* and its class an *aggregated class*.

composition

An object may be owned by several other aggregating objects. If an object is exclusively owned by an aggregating object, the relationship between them is referred to as *composition*. For example, "a student has a name" is a composition relationship between the `Student` class and the `Name` class, whereas "a student has an address" is an aggregation relationship between the `Student` class and the `Address` class, since an address may be shared by several students. In UML, a filled diamond is attached to an aggregating class (e.g., `Student`) to denote the composition relationship with an aggregated class (e.g., `Name`), and an empty diamond is attached to an aggregating class (e.g., `Student`) to denote the aggregation relationship with an aggregated class (e.g., `Address`), as shown in Figure 10.6.

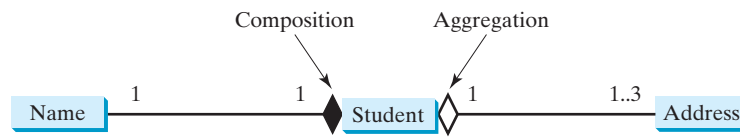


FIGURE 10.6 A student has a name and an address.

multiplicity

Each class involved in a relationship may specify a *multiplicity*. A multiplicity could be a number or an interval that specifies how many objects of the class are involved in the relationship. The character `*` means an unlimited number of objects, and the interval `m..n` means that the number of objects should be between `m` and `n`, inclusive. In Figure 10.6, each student has only one address, and each address may be shared by up to `3` students. Each student has one name, and a name is unique for each student.

An aggregation relationship is usually represented as a data field in the aggregating class. For example, the relationship in Figure 10.6 can be represented as follows:

```
public class Name {
  ...
}
```

Aggregated class

```
public class Student {
  private Name name;
  private Address address;

  ...
}
```

Aggregating class

```
public class Address {
  ...
}
```

Aggregated class

Aggregation may exist between objects of the same class. For example, a person may have a supervisor. This is illustrated in Figure 10.7.

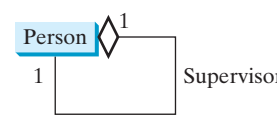


FIGURE 10.7 A person may have a supervisor.

In the relationship "a person has a supervisor," as shown in Figure 10.7, a supervisor can be represented as a data field in the `Person` class, as follows:

```
public class Person {
  // The type for the data is the class itself
```

```
    private Person supervisor;

    ...
}
```

If a person may have several supervisors, as shown in Figure 10.8(a), you may use an array to store supervisors, as shown in Figure 10.8(b).

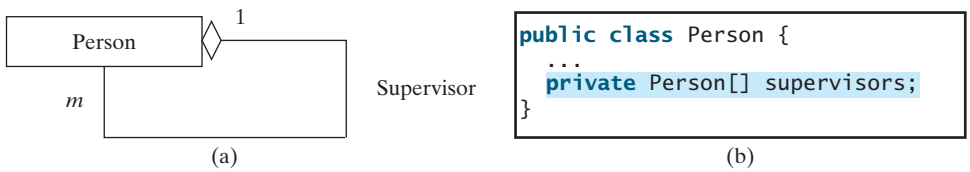


FIGURE 10.8 A person may have several supervisors.



Note

Since aggregation and composition relationships are represented using classes in similar ways, many texts don't differentiate them and call both compositions.

aggregation or composition

10.8 Designing the **Course** Class

This book's philosophy is *teaching by example and learning by doing*. The book provides a wide variety of examples to demonstrate object-oriented programming. The next three sections offer additional examples on designing classes.

Suppose you need to process course information. Each course has a name and has students enrolled. You should be able to add/drop a student to/from the course. You can use a class to model the courses, as shown in Figure 10.9.

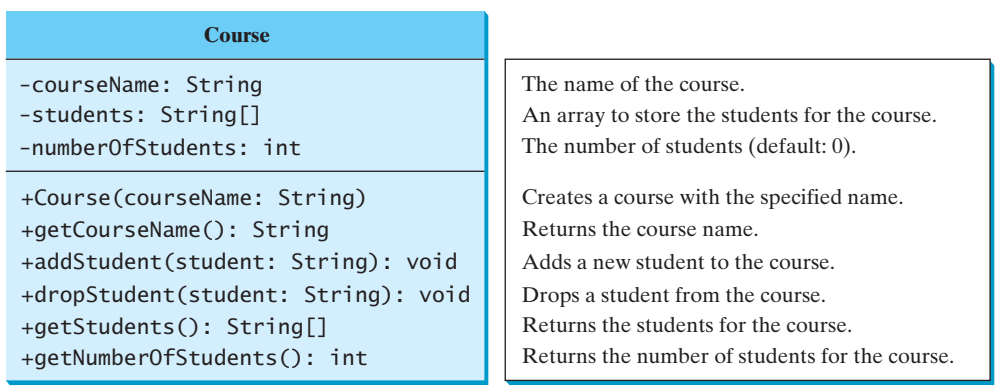

Course
-courseName: String
-students: String[]
-numberOfStudents: int
+Course(courseName: String)
+getCourseName(): String
+addStudent(student: String): void
+dropStudent(student: String): void
+getStudents(): String[]
+getNumberOfStudents(): int
The name of the course.
An array to store the students for the course.
The number of students (default: 0).
Creates a course with the specified name.
Returns the course name.
Adds a new student to the course.
Drops a student from the course.
Returns the students for the course.
Returns the number of students for the course.

FIGURE 10.9 The `Course` class models the courses.

A `Course` object can be created using the constructor `Course(String name)` by passing a course name. You can add students to the course using the `addStudent(String student)` method, drop a student from the course using the `dropStudent(String student)` method, and return all the students for the course using the `getStudents()` method. Suppose the class is available; Listing 10.5 gives a test class that creates two courses and adds students to them.

LISTING 10.5 TestCourse.java

create a course
add a student
number of students
return students

```
1 public class TestCourse {
2   public static void main(String[] args) {
3     Course course1 = new Course("Data Structures");
4     Course course2 = new Course("Database Systems");
5
6     course1.addStudent("Peter Jones");
7     course1.addStudent("Brian Smith");
8     course1.addStudent("Anne Kennedy");
9
10    course2.addStudent("Peter Jones");
11    course2.addStudent("Steve Smith");
12
13    System.out.println("Number of students in course1: "
14      + course1.getNumberOfStudents());
15    String[] students = course1.getStudents();
16    for (int i = 0; i < course1.getNumberOfStudents(); i++)
17      System.out.print(students[i] + ", ");
18
19    System.out.println();
20    System.out.print("Number of students in course2: "
21      + course2.getNumberOfStudents());
22  }
23 }
```



```
Number of students in course1: 3
Peter Jones, Brian Smith, Anne Kennedy,
Number of students in course2: 2
```

The **Course** class is implemented in Listing 10.6. It uses an array to store the students for the course. For simplicity, assume that the maximum course enrollment is **100**. The array is created using **new String[100]** in line 3. The **addStudent** method (line 10) adds a student to the array. Whenever a new student is added to the course, **numberOfStudents** is increased (line 12). The **getStudents** method returns the array. The **dropStudent** method (line 27) is left as an exercise.

LISTING 10.6 Course.java

create students
add a course
return students

```
1 public class Course {
2   private String courseName;
3   private String[] students = new String[100];
4   private int numberOfStudents;
5
6   public Course(String courseName) {
7     this.courseName = courseName;
8   }
9
10  public void addStudent(String student) {
11    students[numberOfStudents] = student;
12    numberOfStudents++;
13  }
14
15  public String[] getStudents() {
16    return students;
17  }
18
```

```
19   public int getNumberOfStudents() {
20     return numberOfStudents;
21   }
22
23   public String getCourseName() {
24     return courseName;
25   }
26
27   public void dropStudent(String student) {
28     // Left as an exercise in Exercise 10.9
29   }
30 }
```

number of students

The array size is fixed to be `100` (line 3) in Listing 10.6. You can improve it to automatically increase the array size in Exercise 10.9.

When you create a `Course` object, an array object is created. A `Course` object contains a reference to the array. For simplicity, you can say that the `Course` object contains the array.

The user can create a `Course` and manipulate it through the public methods `addStudent`, `dropStudent`, `getNumberOfStudents`, and `getStudents`. However, the user doesn't need to know how these methods are implemented. The `Course` class encapsulates the internal implementation. This example uses an array to store students. You may use a different data structure to store students. The program that uses `Course` does not need to change as long as the contract of the public methods remains unchanged.

10.9 Designing a Class for Stacks

Recall that a stack is a data structure that holds data in a last-in, first-out fashion, as shown in Figure 10.10.

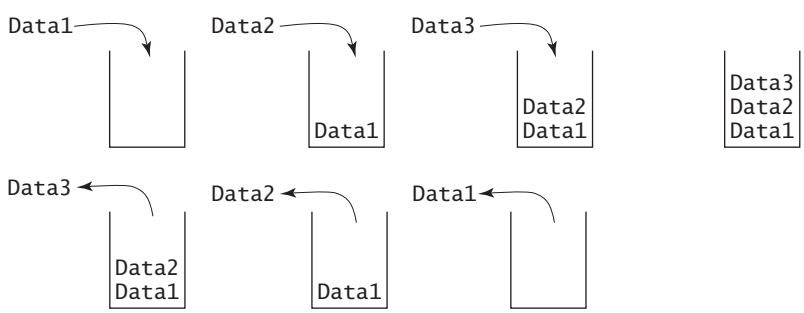


FIGURE 10.10 A stack holds data in a last-in, first-out fashion.

stack

Stacks have many applications. For example, the compiler uses a stack to process method invocations. When a method is invoked, its parameters and local variables are pushed into a stack. When a method calls another method, the new method's parameters and local variables are pushed into the stack. When a method finishes its work and returns to its caller, its associated space is released from the stack.

You can define a class to model stacks. For simplicity, assume the stack holds the `int` values. So, name the stack class `StackOfIntegers`. The UML diagram for the class is shown in Figure 10.11.

Suppose that the class is available. Let us write a test program in Listing 10.7 that uses the class to create a stack (line 3), stores ten integers `0`, `1`, `2`, . . . , and `9` (line 6), and displays them in reverse order (line 9).



Video Note
The `StackOfInteger` class

StackOfIntegers	
-elements: int[]	An array to store integers in the stack.
-size: int	The number of integers in the stack.
+StackOfIntegers()	Constructs an empty stack with a default capacity of 16.
+StackOfIntegers(capacity: int)	Constructs an empty stack with a specified capacity.
+empty(): boolean	Returns true if the stack is empty.
+peek(): int	Returns the integer at the top of the stack without removing it from the stack.
+push(value: int): void	Stores an integer into the top of the stack.
+pop(): int	Removes the integer at the top of the stack and returns it.
+getSize(): int	Returns the number of elements in the stack.

FIGURE 10.11 The `StackOfIntegers` class encapsulates the stack storage and provides the operations for manipulating the stack.

LISTING 10.7 TestStackOfIntegers.java

```
1  public class TestStackOfIntegers {
2    public static void main(String[] args) {
3      StackOfIntegers stack = new StackOfIntegers();
4
5      for (int i = 0; i < 10; i++)
6        stack.push(i);
7
8      while (!stack.empty())
9        System.out.print(stack.pop() + " ");
10   }
11 }
```

create a stack

push to stack

pop from stack



```
9 8 7 6 5 4 3 2 1 0
```

How do you implement the `StackOfIntegers` class? The elements in the stack are stored in an array named `elements`. When you create a stack, the array is also created. The no-arg constructor creates an array with the default capacity of `16`. The variable `size` counts the number of elements in the stack, and `size - 1` is the index of the element at the top of the stack, as shown in Figure 10.12. For an empty stack, `size` is `0`.

The `StackOfIntegers` class is implemented in Listing 10.8. The methods `empty()`, `peek()`, `pop()`, and `getSize()` are easy to implement. To implement `push(int value)`, assign `value` to `elements[size]` if `size < capacity` (line 24). If the stack is full (i.e.,

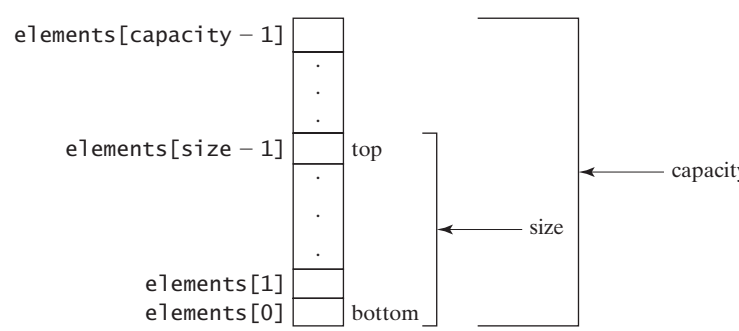


FIGURE 10.12 The `StackOfIntegers` class encapsulates the stack storage and provides the operations for manipulating the stack.

`size >= capacity`), create a new array of twice the current capacity (line 19), copy the contents of the current array to the new array (line 20), and assign the reference of the new array to the current array in the stack (line 21). Now you can add the new value to the array (line 24).

LISTING 10.8 StackOfIntegers.java

```
1  public class StackOfIntegers {
2    private int[] elements;
3    private int size;
4    public static final int DEFAULT_CAPACITY = 16;                    max capacity 16
5
6    /** Construct a stack with the default capacity 16 */
7    public StackOfIntegers() {
8      this(DEFAULT_CAPACITY);
9    }
10
11   /** Construct a stack with the specified maximum capacity */
12   public StackOfIntegers(int capacity) {
13     elements = new int[capacity];
14   }
15
16   /** Push a new integer into the top of the stack */
17   public void push(int value) {
18     if (size >= elements.length) {
19       int[] temp = new int[elements.length * 2];                    double the capacity
20       System.arraycopy(elements, 0, temp, 0, elements.length);
21       elements = temp;
22     }
23
24     elements[size++] = value;                                       add to stack
25   }
26
27   /** Return and remove the top element from the stack */
28   public int pop() {
29     return elements[--size];
30   }
31
32   /** Return the top element from the stack */
33   public int peek() {
34     return elements[size - 1];
35   }
36
37   /** Test whether the stack is empty */
38   public boolean empty() {
39     return size == 0;
40   }
41
42   /** Return the number of elements in the stack */
43   public int getSize() {
44     return size;
45   }
46 }
```

10.10 Designing the **GuessDate** Class

Listing 3.3, GuessBirthday.java, and Listing 7.6, GuessBirthdayUsingArray.java, presented two programs for guessing birthdays. Both programs use the same data developed with the procedural paradigm. The majority of code in these two programs is to define the five sets of data. You cannot reuse the code in these two programs. To make the code reusable, design a class to encapsulate the data, as defined in Figure 10.13.

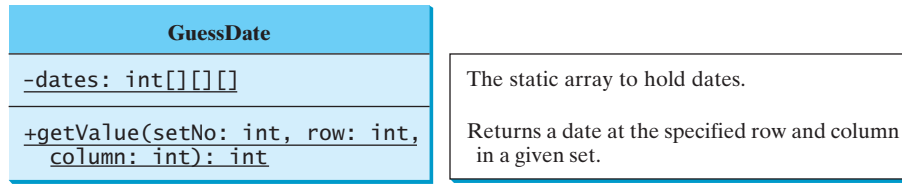

GuessDate
-dates: int[][][]
+getValue(setNo: int, row: int, column: int): int
The static array to hold dates.
Returns a date at the specified row and column in a given set.

FIGURE 10.13 The `GuessDate` class defines data for guessing birthdays.

Note that `getValue` is defined as a static method because it is not dependent on a specific object of the `GuessDate` class. The `GuessDate` class encapsulates `dates` as a private member. The user of this class need not know how `dates` is implemented or even that the `dates` field exists in the class. All the user needs to know is how to use this method to access dates. Suppose this class is available. As shown in §3.5, there are five sets of dates. Invoking `getValue(setNo, row, column)` returns the date at the specified row and column in the given set. For example, `getValue(1, 0, 0)` returns `2`.

Assume that the `GuessDate` class is available. Listing 10.9 is a test program that uses this class.

LISTING 10.9 UseGuessDateClass.java

```
1 import java.util.Scanner;
2
3 public class UseGuessDateClass {
4   public static void main(String[] args) {
5     int date = 0; // Date to be determined
6     int answer;
7
8     // Create a Scanner
9     Scanner input = new Scanner(System.in);
10
11    for (int i = 0; i < 5; i++) {
12      System.out.println("Is your birthday in Set" + (i + 1) + "?");
13      for (int j = 0; j < 4; j++) {
14        for (int k = 0; k < 4; k++)
15          System.out.print(GuessDate.getValue(i, j, k) + "  ");
16        System.out.println();
17      }
18
19      System.out.print("\nEnter 0 for No and 1 for Yes: ");
20      answer = input.nextInt();
21
22      if (answer == 1)
23        date += GuessDate.getValue(i, 0, 0);
24    }
25
26    System.out.println("Your birthday is " + date);
27  }
28 }
```

invoke static method

invoke static method



```
Is your birthday in Set1?
1  3  5  7
9  11  13  15
17  19  21  23
25  27  29  31
Enter 0 for No and 1 for Yes: 0 Enter
```

```
Is your birthday in Set2?
2  3  6  7
10  11  14  15
18  19  22  23
26  27  30  31
Enter 0 for No and 1 for Yes: 1 Enter

Is your birthday in Set3?
4  5  6  7
12  13  14  15
20  21  22  23
28  29  30  31
Enter 0 for No and 1 for Yes: 0 Enter

Is your birthday in Set4?
8  9  10  11
12  13  14  15
24  25  26  27
28  29  30  31
Enter 0 for No and 1 for Yes: 1 Enter

Is your birthday in Set5?
16  17  18  19
20  21  22  23
24  25  26  27
28  29  30  31
Enter 0 for No and 1 for Yes: 1 Enter

Your birthday is 26
```

Since `getValue` is a static method, you don't need to create an object in order to invoke it. `GuessDate.getValue(i, j, k)` (line 15) returns the date at row `i` and column `k` in Set `i`.

The `GuessDate` class can be implemented in Listing 10.10.

LISTING 10.10 GuessDate.java

```
 1 public class GuessDate {
 2   private final static int[][][] dates = {                    static field
 3     {{ 1,  3,  5,  7},
 4      { 9, 11, 13, 15},
 5      {17, 19, 21, 23},
 6      {25, 27, 29, 31}},
 7     {{ 2,  3,  6,  7},
 8      {10, 11, 14, 15},
 9      {18, 19, 22, 23},
10      {26, 27, 30, 31}},
11     {{ 4,  5,  6,  7},
12      {12, 13, 14, 15},
13      {20, 21, 22, 23},
14      {28, 29, 30, 31}},
15     {{ 8,  9, 10, 11},
16      {12, 13, 14, 15},
17      {24, 25, 26, 27},
18      {28, 29, 30, 31}},
19     {{16, 17, 18, 19},
20      {20, 21, 22, 23},
21      {24, 25, 26, 27},
22      {28, 29, 30, 31}}};
```

```
23
24   /** Prevent the user from creating objects from GuessDate */
25   private GuessDate() {
26   }
27
28   /** Return a date at the specified row and column in a given set */
29   public static int getValue(int setNo, int k, int j) {
30     return dates[setNo][k][j];
31   }
32 }
```

private constructor

static method

This class uses a three-dimensional array to store dates (lines 2–22). You may use a different data structure (i.e., five two-dimensional arrays for representing five sets of numbers). The implementation of the `getValue` method will change, but the program that uses `GuessDate` does not need to change as long as the contract of the public method `getValue` remains unchanged. This shows the benefit of data encapsulation.

benefit of data encapsulation

private constructor

The class defines a private no-arg constructor (line 25) to prevent the user from creating objects for this class. Since all methods are static in this class, there is no need to create objects from this class.

10.11 Class Design Guidelines

You have learned how to design classes from the preceding two examples and from many other examples in the preceding chapters. Here are some guidelines.

10.11.1 Cohesion

coherent purpose

A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose. You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff are different entities.

separating responsibilities

A single entity with too many responsibilities can be broken into several classes to separate responsibilities. The classes `String`, `StringBuilder`, and `StringBuffer` all deal with strings, for example, but have different responsibilities. The `String` class deals with immutable strings, the `StringBuilder` class is for creating mutable strings, and the `StringBuffer` class is similar to `StringBuilder` except that `StringBuffer` contains synchronized methods for updating strings.

10.11.2 Consistency

naming conventions

Follow standard Java programming style and naming conventions. Choose informative names for classes, data fields, and methods. A popular style is to place the data declaration before the constructor and place constructors before methods.

naming consistency

Choose names consistently. It is not a good practice to choose different names for similar operations. For example, the `length()` method returns the size of a `String`, a `StringBuilder`, and a `StringBuffer`. It would be inconsistent if different names were used for this method in these classes.

no-arg constructor

In general, you should consistently provide a public no-arg constructor for constructing a default instance. If a class does not support a no-arg constructor, document the reason. If no constructors are defined explicitly, a public default no-arg constructor with an empty body is assumed.

If you want to prevent users from creating an object for a class, you may declare a private constructor in the class, as is the case for the `Math` class and the `GuessDate` class.

10.11.3 Encapsulation

encapsulating data fields

A class should use the `private` modifier to hide its data from direct access by clients. This makes the class easy to maintain.

Provide a `get` method only if you want the field to be readable, and provide a `set` method only if you want the field to be updateable. For example, the `Course` class provides a `get` method for `courseName`, but no `set` method, because the user is not allowed to change the course name, once it is created.

10.11.4 Clarity

easy to explain

Cohesion, consistency, and encapsulation are good guidelines for achieving design clarity. Additionally, a class should have a clear contract that is easy to explain and easy to understand.

Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on what or when the user can do with it, design the properties in a way that lets the user set them in any order and with any combination of values, and design methods that function independently of their order of occurrence. For example, the `Loan` class contains the properties `loanAmount`, `numberOfYears`, and `annualInterestRate`. The values of these properties can be set in any order.

independent methods

intuitive meaning

Methods should be defined intuitively without generating confusion. For example, the `substring(int beginIndex, int endIndex)` method in the `String` class is somehow confusing. The method returns a substring from `beginIndex` to `endIndex - 1`, rather than `endIndex`.

independent properties

You should not declare a data field that can be derived from other data fields. For example, the following `Person` class has two data fields: `birthDate` and `age`. Since `age` can be derived from `birthDate`, age should not be declared as a data field.

```
public class Person {
  private java.util.Date birthDate;
  private int age;

  ...
}
```

10.11.5 Completeness

Classes are designed for use by many different customers. In order to be useful in a wide range of applications, a class should provide a variety of ways for customization through properties and methods. For example, the `String` class contains more than 40 methods that are useful for a variety of applications.

10.11.6 Instance vs. Static

A variable or method that is dependent on a specific instance of the class should be an instance variable or method. A variable that is shared by all the instances of a class should be declared static. For example, the variable `numberOfObjects` in `Circle3` in Listing 8.9, is shared by all the objects of the `SimpleCircle1` class and therefore is declared static. A method that is not dependent on a specific instance should be declared as a static method. For instance, the `getNumberOfObjects` method in `Circle3` is not tied to any specific instance and therefore is declared as a static method.

Always reference static variables and methods from a class name (rather than a reference variable) to improve readability and avoid errors.

Do not pass a parameter from a constructor to initialize a static data field. It is better to use a `set` method to change the static data field. The class in (a) below is better replaced by (b).

```
public class SomeThing {
  private int t1;
  private static int t2;

  public SomeThing(int t1, int t2) {
    ...
  }
}
```

(a)

```
public class SomeThing {
  private int t1;
  private static int t2;

  public SomeThing(int t1) {
    ...
  }

  public static void setT2(int t2) {
    SomeThing.t2 = t2;
  }
}
```

(b)

common design error

Instance and static are integral parts of object-oriented programming. A data field or method is either instance or static. Do not mistakenly overlook static data fields or methods. It is a common design error to define an instance method that should have been static. For example, the `factorial(int n)` method for computing the factorial of `n` should be defined static, because it is independent of any specific instance.

A constructor is always instance, because it is used to create a specific instance. A static variable or method can be invoked from an instance method, but an instance variable or method cannot be invoked from a static method.

Key Terms

class abstraction 347
class encapsulation 347
class's contract 347
class's variable 345
immutable class 344
immutable object 344
stack 357
`this` keyword 346

Chapter Summary

1. Once it is created, an immutable object cannot be modified. To prevent users from modifying an object, you may define immutable classes.

2. The scope of instance and static variables is the entire class, regardless of where the variables are declared. Instance and static variables can be declared anywhere in the class. For consistency, they are declared at the beginning of the class.

3. The keyword `this` can be used to refer to the calling object. It can also be used inside a constructor to invoke another constructor of the same class.

4. The procedural paradigm focuses on designing methods. The object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

Review Questions

Section 10.2

10.1 If a class contains only private data fields and no set methods, is the class immutable?

10.2 If all the data fields in a class are private and primitive type, and the class contains no `set` methods, is the class immutable?

10.3 Is the following class immutable?

```
public class A {
  private int[] values;

  public int[] getValues() {
    return values;
  }
}
```

10.4 If you redefine the `Loan` class in Listing 10.2 without `set` methods, is the class immutable?

Section 10.3

10.5 What is the output of the following program?

```
public class Foo {
  private static int i = 0;
  private static int j = 0;

  public static void main(String[] args) {
    int i = 2;
    int k = 3;

    {
      int j = 3;
      System.out.println("i + j is " + i + j);
    }

    k = i + j;
    System.out.println("k is " + k);
    System.out.println("j is " + j);
  }
}
```

Section 10.4

10.6 Describe the role of the `this` keyword. What is wrong in the following code?

```
 1 public class C {
 2   private int p;
 3
 4   public C() {
 5     System.out.println("C's no-arg constructor invoked");
 6     this(0);
 7   }
 8
 9   public C(int p) {
10     p = p;
11   }
12
13   public void setP(int p) {
14     p = p;
15   }
16 }
```

PROGRAMMING EXERCISES

10.1* (*The `Time` class*) Design a class named `Time`. The class contains:

- Data fields `hour`, `minute`, and `second` that represent a time.
- A no-arg constructor that creates a `Time` object for the current time. (The values of the data fields will represent the current time.)
- A constructor that constructs a `Time` object with a specified elapsed time since midnight, Jan 1, 1970, in milliseconds. (The values of the data fields will represent this time.)
- A constructor that constructs a `Time` object with the specified hour, minute, and second.
- Three `get` methods for the data fields `hour`, `minute`, and `second`, respectively.
- A method named `setTime(long elapseTime)` that sets a new time for the object using the elapsed time.

Draw the UML diagram for the class. Implement the class. Write a test program that creates two `Time` objects (using `new Time()` and `new Time(555550000)`) and display their hour, minute, and second.

(*Hint:* The first two constructors will extract hour, minute, and second from the elapsed time. For example, if the elapsed time is `555550` seconds, the hour is `10`, the minute is `19`, and the second is `9`. For the no-arg constructor, the current time can be obtained using `System.currentTimeMills()`, as shown in Listing 2.6, ShowCurrentTime.java.)

10.2 (*The `BMI` class*) Add the following new constructor in the `BMI` class:

```
/** Construct a BMI with the specified name, age, weight,
 * feet and inches
 */
public BMI(String name, int age, double weight, double feet,
double inches)
```

10.3 (*The `MyInteger` class*) Design a class named `MyInteger`. The class contains:

- An `int` data field named `value` that stores the `int` value represented by this object.
- A constructor that creates a `MyInteger` object for the specified `int` value.
- A `get` method that returns the `int` value.
- Methods `isEven()`, `isOdd()`, and `isPrime()` that return `true` if the value is even, odd, or prime, respectively.
- Static methods `isEven(int)`, `isOdd(int)`, and `isPrime(int)` that return `true` if the specified value is even, odd, or prime, respectively.
- Static methods `isEven(MyInteger)`, `isOdd(MyInteger)`, and `isPrime(MyInteger)` that return `true` if the specified value is even, odd, or prime, respectively.
- Methods `equals(int)` and `equals(MyInteger)` that return `true` if the value in the object is equal to the specified value.
- A static method `parseInt(char[])` that converts an array of numeric characters to an `int` value.
- A static method `parseInt(String)` that converts a string into an `int` value.

Draw the UML diagram for the class. Implement the class. Write a client program that tests all methods in the class.

10.4 (*The `MyPoint` class*) Design a class named `MyPoint` to represent a point with `x`- and `y`-coordinates. The class contains:



Video Note
The `MyPoint` class

- Two data fields `x` and `y` that represent the coordinates with get methods.
- A no-arg constructor that creates a point (`0`, `0`).
- A constructor that constructs a point with specified coordinates.
- Two `get` methods for data fields `x` and `y`, respectively.
- A method named `distance` that returns the distance from this point to another point of the `MyPoint` type.
- A method named `distance` that returns the distance from this point to another point with specified `x`- and `y`-coordinates.

Draw the UML diagram for the class. Implement the class. Write a test program that creates two points (`0`, `0`) and (`10`, `30.5`) and displays the distance between them.

10.5* (*Displaying the prime factors*) Write a program that prompts the user to enter a positive integer and displays all its smallest factors in decreasing order. For example, if the integer is `120`, the smallest factors are displayed as `5`, `3`, `2`, `2`, `2`. Use the `StackOfIntegers` class to store the factors (e.g., `2`, `2`, `2`, `3`, `5`) and retrieve and display them in reverse order.

10.6* (*Displaying the prime numbers*) Write a program that displays all the prime numbers less than `120` in decreasing order. Use the `StackOfIntegers` class to store the prime numbers (e.g., `2`, `3`, `5`, ...) and retrieve and display them in reverse order.

10.7** (*Game: ATM machine*) Use the `Account` class created in Exercise 8.7 to simulate an ATM machine. Create ten accounts in an array with id `0`, `1`, ..., `9`, and initial balance $100. The system prompts the user to enter an id. If the id is entered incorrectly, ask the user to enter a correct id. Once an id is accepted, the main menu is displayed as shown in the sample run. You can enter a choice `1` for viewing the current balance, `2` for withdrawing money, `3` for depositing money, and `4` for exiting the main menu. Once you exit, the system will prompt for an id again. So, once the system starts, it will not stop.



```
Enter an id: 4 Enter

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 1 Enter
The balance is 100.0

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 2 Enter
Enter an amount to withdraw: 3 Enter
```

```
Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 1 Enter
The balance is 97.0

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 3 Enter
Enter an amount to deposit: 10 Enter

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 1 Enter
The balance is 107.0

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 4 Enter

Enter an id:
```

10.8** (*Financial: the `Tax` class*) Exercise 7.12 writes a program for computing taxes using arrays. Design a class named `Tax` to contain the following instance data fields:

- `int filingStatus`: One of the four tax-filing statuses: `0`—single filer, `1`—married filing jointly, `2`—married filing separately, and `3`—head of household. Use the public static constants `SINGLE_FILER` (`0`), `MARRIED_JOINTLY` (`1`), `MARRIED_SEPARATELY` (`2`), `HEAD_OF_HOUSEHOLD` (`3`) to represent the status.
- `int[][] brackets`: Stores the tax brackets for each filing status.
- `double[] rates`: Stores the tax rates for each bracket.
- `double taxableIncome`: Stores the taxable income.

Provide the `get` and `set` methods for each data field and the `getTax()` method that returns the tax. Also provide a no-arg constructor and the constructor `Tax(filingStatus, brackets, rates, taxableIncome)`.

Draw the UML diagram for the class. Implement the class. Write a test program that uses the `Tax` class to print the 2001 and 2009 tax tables for taxable income from $50,000 to $60,000 with intervals of $1,000 for all four statuses. The tax rates for the year 2009 were given in Table 3.2. The tax rates for 2001 are shown in Table 10.1.

TABLE 10.1 2001 United States Federal Personal Tax Rates

Tax rate	*Single filers*	*Married filing jointly or qualifying widow(er)*	*Married filing separately*	*Head of household*
15%	Up to $27,050	Up to $45,200	UP to $22,600	UP to $36,250
27.5%	$27,051–$65,550	$45,201–$109,250	$22,601–$54,625	$36,251–$93,650
30.5%	$65,551–$136,750	$109,251–$166,500	$54,626–$83,250	$93,651–$151,650
35.5%	$136,751–$297,350	$166,501–$297,350	$83,251–148,675	$151,651–$297,350
39.1%	$297,351 or more	$297,351 or more	$148,676 or more	$297,351 or more

10.9** (*The `Course` class*) Revise the `Course` class as follows:

- The array size is fixed in Listing 10.6. Improve it to automatically increase the array size by creating a new larger array and copying the contents of the current array to it.
- Implement the `dropStudent` method.
- Add a new method named `clear()` that removes all students from the course.

Write a test program that creates a course, adds three students, removes one, and displays the students in the course.

10.10* (*Game: The `GuessDate` class*) Modify the `GuessDate` class in Listing 10.10. Instead of representing dates in a three-dimensional array, use five two-dimensional arrays to represent the five sets of numbers. So you need to declare:

```
private static int[][] set1 = {{1,  3,  5,  7}, ... };
private static int[][] set2 = {{2,  3,  6,  7}, ... };
private static int[][] set3 = {{4,  5,  6,  7}, ... };
private static int[][] set4 = {{8,  9, 10, 11}, ... };
private static int[][] set5 = {{16, 17, 18, 19}, ... };
```

10.11* (*Geometry: The `Circle2D` class*) Define the `Circle2D` class that contains:

- Two `double` data fields named `x` and `y` that specify the center of the circle with `get` methods.
- A data field `radius` with a `get` method.
- A no-arg constructor that creates a default circle with (`0`, `0`) for (`x`, `y`) and `1` for `radius`.
- A constructor that creates a circle with the specified `x`, `y`, and `radius`.
- A method `getArea()` that returns the area of the circle.
- A method `getPerimeter()` that returns the perimeter of the circle.
- A method `contains(double x, double y)` that returns `true` if the specified point (`x`, `y`) is inside this circle. See Figure 10.14(a).
- A method `contains(Circle2D circle)` that returns `true` if the specified circle is inside this circle. See Figure 10.14(b).
- A method `overlaps(Circle2D circle)` that returns `true` if the specified circle overlaps with this circle. See Figure 10.14(c).

Draw the UML diagram for the class. Implement the class. Write a test program that creates a `Circle2D` object `c1` (`new Circle2D(2, 2, 5.5)`), displays its area and perimeter, and displays the result of `c1.contains(3, 3)`, `c1.contains(new Circle2D(4, 5, 10.5))`, and `c1.overlaps(new Circle2D(3, 5, 2.3))`.

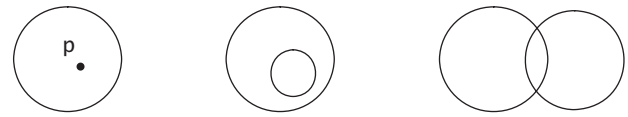


FIGURE 10.14 (a) A point is inside the circle. (b) A circle is inside another circle. (c) A circle overlaps another circle.

10.12* (*Geometry: The `MyRectangle2D` class*) Define the `MyRectangle2D` class that contains:

- Two `double` data fields named `x` and `y` that specify the center of the rectangle with `get` and `set` methods. (Assume that the rectangle sides are parallel to `x-` or `y-` axes.)
- The data fields `width` and `height` with `get` and `set` methods.
- A no-arg constructor that creates a default rectangle with (`0`, `0`) for (`x`, `y`) and `1` for both `width` and `height`.
- A constructor that creates a rectangle with the specified `x`, `y`, `width`, and `height`.
- A method `getArea()` that returns the area of the rectangle.
- A method `getPerimeter()` that returns the perimeter of the rectangle.
- A method `contains(double x, double y)` that returns `true` if the specified point (`x`, `y`) is inside this rectangle. See Figure 10.15(a).
- A method `contains(MyRectangle2D r)` that returns `true` if the specified rectangle is inside this rectangle. See Figure 10.15(b).
- A method `overlaps(MyRectangle2D r)` that returns `true` if the specified rectangle overlaps with this rectangle. See Figure 10.15(c).

Draw the UML diagram for the class. Implement the class. Write a test program that creates a `MyRectangle2D` object `r1` (`new MyRectangle2D(2, 2, 5.5, 4.9)`), displays its area and perimeter, and displays the result of `r1.contains(3, 3)`, `r1.contains(new MyRectangle2D(4, 5, 10.5, 3.2))`, and `r1.overlaps(new MyRectangle2D(3, 5, 2.3, 5.4))`.

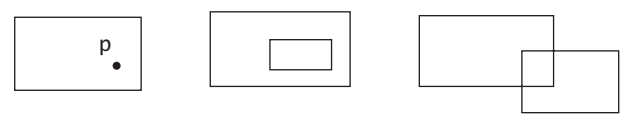


FIGURE 10.15 (a) A point is inside the rectangle. (b) A rectangle is inside another rectangle. (c) A rectangle overlaps another rectangle.

10.13*** (*Geometry: The `Triangle2D` class*) Define the `Triangle2D` class that contains:

- Three points named `p1`, `p2`, and `p3` with the type `MyPoint` with `get` and `set` methods. `MyPoint` is defined in Exercise 10.4.
- A no-arg constructor that creates a default triangle with points (`0`, `0`), (`1`, `1`), and (`2`, `5`).
- A constructor that creates a triangle with the specified points.
- A method `getArea()` that returns the area of the triangle.
- A method `getPerimeter()` that returns the perimeter of the triangle.

- A method `contains(MyPoint p)` that returns `true` if the specified point `p` is inside this triangle. See Figure 10.16(a).
- A method `contains(Triangle2D t)` that returns `true` if the specified triangle is inside this triangle. See Figure 10.16(b).
- A method `overlaps(Triangle2D t)` that returns `true` if the specified triangle overlaps with this triangle. See Figure 10.16(c).

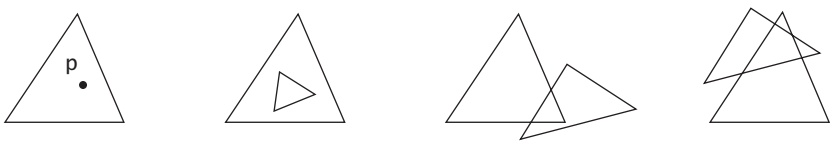


FIGURE 10.16 (a) A point is inside the triangle. (b) A triangle is inside another triangle. (c) A triangle overlaps another triangle.

Draw the UML diagram for the class. Implement the class. Write a test program that creates a `Triangle2D` objects `t1` (`new Triangle2D(new MyPoint(2.5, 2), new MyPoint(4.2, 3), MyPoint(5, 3.5))`), displays its area and perimeter, and displays the result of `t1.contains(3, 3)`, `r1.contains(new Triangle2D(new MyPoint(2.9, 2), new MyPoint(4, 1), MyPoint(1, 3.4)))`, and `t1.overlaps(new Triangle2D(new MyPoint(2, 5.5), new MyPoint(4, -3), MyPoint(2, 6.5)))`.

(*Hint*: For the formula to compute the area of a triangle, see Exercise 5.19. Use the `java.awt.geo.Line2D` class in the Java API to implement the `contains` and `overlaps` methods. The `Line2D` class contains the methods for checking whether two line segments intersect and whether a line contains a point, etc. Please see the Java API for more information on `Line2D`. To detect whether a point is inside a triangle, draw three dashed lines, as shown in Figure 10.17. If the point is inside a triangle, each dashed line should intersect a side only once. If a dashed line intersects a side twice, then the point must be outside the triangle.)



FIGURE 10.17 (a) A point is inside the triangle. (b) A point is outside the triangle.

10.14* (*The `MyDate` class*) Design a class named `MyDate`. The class contains:

- Data fields `year`, `month`, and `day` that represent a date.
- A no-arg constructor that creates a `MyDate` object for the current date.
- A constructor that constructs a `MyDate` object with a specified elapsed time since midnight, Jan 1, 1970, in milliseconds.
- A constructor that constructs a `MyDate` object with the specified year, month, and day.
- Three `get` methods for the data fields `year`, `month`, and `day`, respectively.
- A method named `setDate(long elapsedTime)` that sets a new date for the object using the elapsed time.

Draw the UML diagram for the class. Implement the class. Write a test program that creates two `Date` objects (using `new Date()` and `new Date(34355555133101L)`) and display their hour, minute, and second.

(*Hint*: The first two constructors will extract year, month, and day from the elapsed time. For example, if the elapsed time is `561555550000` milliseconds, the year is `1987`, the month is `10`, and the day is `17`. For the no-arg constructor, the current time can be obtained using `System.currentTimeMills()`, as shown in Listing 2.6, ShowCurrentTime.java.)

CHAPTER 11

INHERITANCE AND POLYMORPHISM

Objectives

- To develop a subclass from a superclass through inheritance (§11.2).
- To invoke the superclass's constructors and methods using the `super` keyword (§11.3).
- To override instance methods in the subclass (§11.4).
- To distinguish differences between overriding and overloading (§11.5).
- To explore the `toString()` method in the `Object` class (§11.6).
- To discover polymorphism and dynamic binding (§§11.7–11.8).
- To describe casting and explain why explicit downcasting is necessary (§11.9).
- To explore the `equals` method in the `Object` class (§11.10).
- To store, retrieve, and manipulate objects in an `ArrayList` (§11.11).
- To implement a `Stack` class using `ArrayList` (§11.12).
- To enable data and methods in a superclass accessible in subclasses using the `protected` visibility modifier (§11.13).
- To prevent class extending and method overriding using the `final` modifier (§11.14).



11.1 Introduction

why inheritance?

Object-oriented programming allows you to derive new classes from existing classes. This is called *inheritance*. Inheritance is an important and powerful feature in Java for reusing software. Suppose you are to define classes to model circles, rectangles, and triangles. These classes have many common features. What is the best way to design these classes so to avoid redundancy and make the system easy to comprehend and easy to maintain? The answer is to use inheritance.

11.2 Superclasses and Subclasses



Video Note
Geometric class hierarchy

You use a class to model objects of the same type. Different classes may have some common properties and behaviors, which can be generalized in a class that can be shared by other classes. Inheritance enables you to define a general class and later extend it to more specialized classes. The specialized classes inherit the properties and methods from the general class.

Consider geometric objects. Suppose you want to design the classes to model geometric objects such as circles and rectangles. Geometric objects have many common properties and behaviors. They can be drawn in a certain color, filled or unfilled. Thus a general class

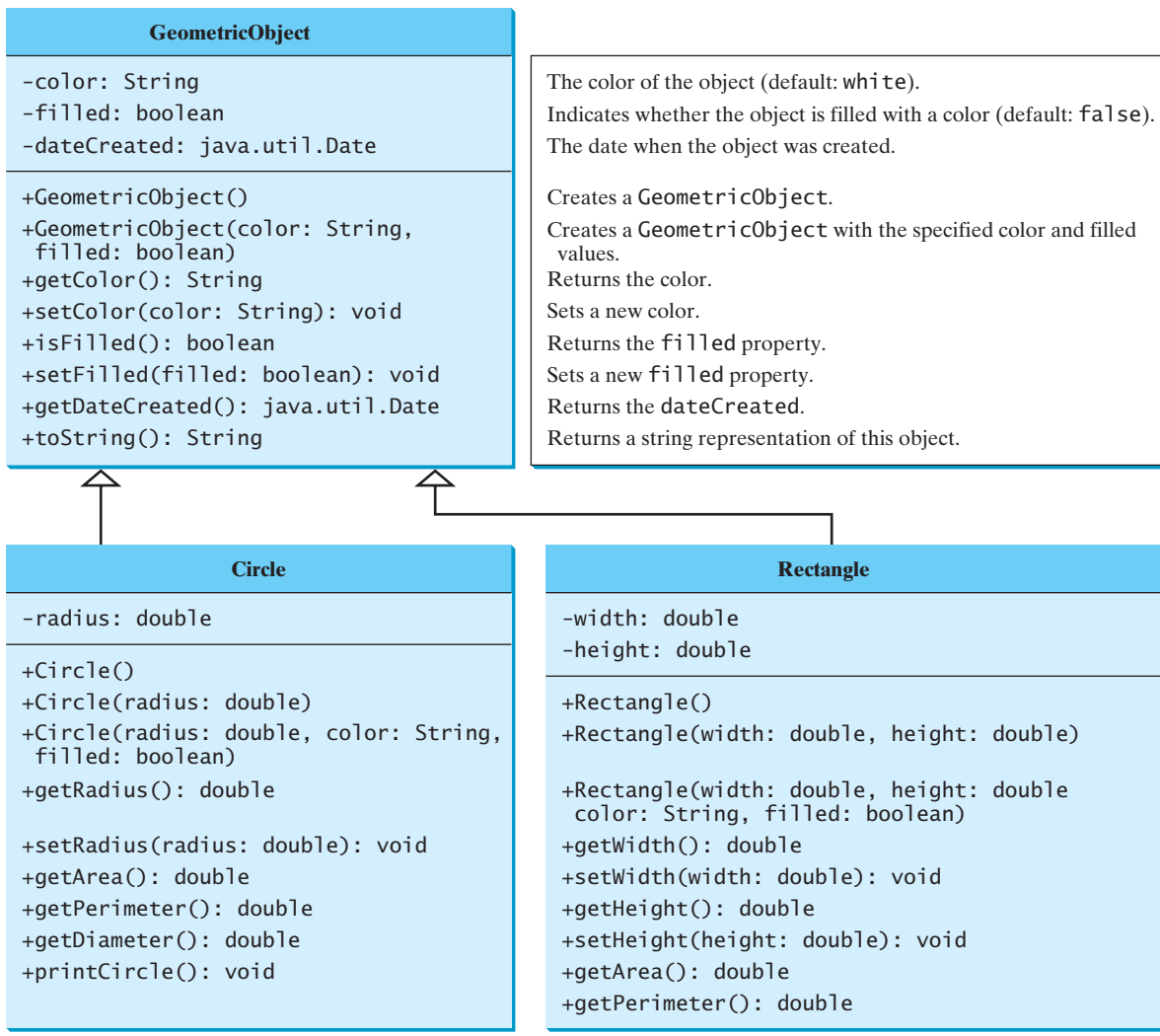


FIGURE 11.1 The `GeometricObject` class is the superclass for `Circle` and `Rectangle`.

`GeometricObject` can be used to model all geometric objects. This class contains the properties `color` and `filled` and their appropriate `get` and `set` methods. Assume that this class also contains the `dateCreated` property and the `getDateCreated()` and `toString()` methods. The `toString()` method returns a string representation for the object. Since a circle is a special type of geometric object, it shares common properties and methods with other geometric objects. Thus it makes sense to define the `Circle` class that extends the `GeometricObject` class. Likewise, `Rectangle` can also be declared as a subclass of `GeometricObject`. Figure 11.1 shows the relationship among these classes. An arrow pointing to the superclass is used to denote the inheritance relationship between the two classes involved.

In Java terminology, a class `C1` extended from another class `C2` is called a *subclass*, and `C2` is called a *superclass*. A superclass is also referred to as a *parent class,* or a *base class*, and a subclass as a *child class*, an *extended class*, or a *derived class*. A subclass inherits accessible data fields and methods from its superclass and may also add new data fields and methods.

subclass

superclass

The `Circle` class inherits all accessible data fields and methods from the `GeometricObject` class. In addition, it has a new data field, `radius`, and its associated `get` and `set` methods. It also contains the `getArea()`, `getPerimeter()`, and `getDiameter()` methods for returning the area, perimeter, and diameter of the circle.

The `Rectangle` class inherits all accessible data fields and methods from the `GeometricObject` class. In addition, it has the data fields `width` and `height` and the associated `get` and `set` methods. It also contains the `getArea()` and `getPerimeter()` methods for returning the area and perimeter of the rectangle.

The `GeometricObject`, `Circle`, and `Rectangle` classes are shown in Listings 11.1, 11.2, and 11.3.



Note

To avoid naming conflict with the improved `GeometricObject`, `Circle`, and `Rectangle` classes introduced in the next chapter, name these classes `GeometricObject1`, `Circle4`, and `Rectangle1` in this chapter. For convenience, we will still refer to them in the text as `GeometricObject`, `Circle`, and `Rectangle` classes. The best way to avoid naming conflict would be to place these classes in a different package. However, for simplicity and consistency, all classes in this book are placed in the default package.

avoid naming conflict

LISTING 11.1 GeometricObject1.java

```
1  public class GeometricObject1 {
2    private String color = "white";
3    private boolean filled;
4    private java.util.Date dateCreated;
5
6    /** Construct a default geometric object */
7    public GeometricObject1() {
8      dateCreated = new java.util.Date();
9    }
10
11   /** Construct a geometric object with the specified color
12     * and filled value */
13   public GeometricObject1(String Color, boolean filled) {
14     dateCreated = new java.util.Date();
15     this.color = color;
16     this.filled = filled;
17   }
18
19   /** Return color */
20   public String getColor() {
21     return color;
22   }
```

data fields

constructor

date constructed

```
23
24   /** Set a new color */
25   public void setColor(String color) {
26     this.color = color;
27   }
28
29   /** Return filled. Since filled is boolean,
30      its get method is named isFilled */
31   public boolean isFilled() {
32     return filled;
33   }
34
35   /** Set a new filled */
36   public void setFilled(boolean filled) {
37     this.filled = filled;
38   }
39
40   /** Get dateCreated */
41   public java.util.Date getDateCreated() {
42     return dateCreated;
43   }
44
45   /** Return a string representation of this object */
46   public String toString() {
47     return "created on " + dateCreated + "\ncolor: " + color +
48       " and filled: " + filled;
49   }
50 }
```

LISTING 11.2 Circle4.java

data fields

constructor

methods

```
1 public class Circle4 extends GeometricObject1 {
2   private double radius;
3
4   public Circle4() {
5   }
6
7   public Circle4(double radius) {
8     this.radius = radius;
9   }
10
11  public Circle4(double radius, String color, boolean filled) {
12    this.radius = radius;
13    setColor(color);
14    setFilled(filled);
15  }
16
17  /** Return radius */
18  public double getRadius() {
19    return radius;
20  }
21
22  /** Set a new radius */
23  public void setRadius(double radius) {
24    this.radius = radius;
25  }
26
27  /** Return area */
28  public double getArea() {
29    return radius * radius * Math.PI;
30  }
```

```
31
32   /** Return diameter */
33   public double getDiameter() {
34     return 2 * radius;
35   }
36
37   /** Return perimeter */
38   public double getPerimeter() {
39     return 2 * radius * Math.PI;
40   }
41
42   /* Print the circle info */
43   public void printCircle() {
44     System.out.println("The circle is created " + getDateCreated() +
45       " and the radius is " + radius);
46   }
47 }
```

The `Circle` class extends the `GeometricObject` class (Listing 11.2) using the following syntax:



The keyword `extends` (line 1) tells the compiler that the `Circle` class extends the `GeometricObject` class, thus inheriting the methods `getColor`, `setColor`, `isFilled`, `setFilled`, and `toString`.

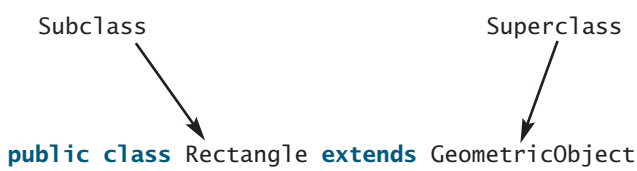
The overloaded constructor `Circle(double radius, string color, boolean filled)` is implemented by invoking the `setColor` and `setFilled` methods to set the `color` and `filled` properties (lines 11–15). These two public methods are defined in the base class `GeometricObject` and are inherited in `Circle`. So, they can be used in the derived class.

You might attempt to use the data fields `color` and `filled` directly in the constructor as follows: private member in base class

```
public Circle4(double radius, String color, boolean filled) {
  this.radius = radius;
  this.color = color; // Illegal
  this.filled = filled; // Illegal
}
```

This is wrong, because the private data fields `color` and `filled` in the `GeometricObject` class cannot be accessed in any class other than in the `GeometricObject` class itself. The only way to read and modify `color` and `filled` is through their `get` and `set` methods.

The `Rectangle` class (Listing 11.3) extends the `GeometricObject` class (Listing 11.2) using the following syntax:



The keyword **extends** (line 1) tells the compiler that the **Rectangle** class extends the **GeometricObject** class, thus inheriting the methods **getColor**, **setColor**, **isFilled**, **setFilled**, and **toString**.

LISTING 11.3 Rectangle1.java

data fields

constructor

methods

```
1  public class Rectangle1 extends GeometricObject1 {
2    private double width;
3    private double height;
4
5    public Rectangle1() {
6    }
7
8    public Rectangle1(double width, double height) {
9      this.width = width;
10     this.height = height;
11   }
12
13   public Rectangle1(double width, double height, String color,
14       boolean filled) {
15     this.width = width;
16     this.height = height;
17     setColor(color);
18     setFilled(filled);
19   }
20
21   /** Return width */
22   public double getWidth() {
23     return width;
24   }
25
26   /** Set a new width */
27   public void setWidth(double width) {
28     this.width = width;
29   }
30
31   /** Return height */
32   public double getHeight() {
33     return height;
34   }
35
36   /** Set a new height */
37   public void setHeight(double height) {
38     this.height = height;
39   }
40
41   /** Return area */
42   public double getArea() {
43     return width * height;
44   }
45
46   /** Return perimeter */
47   public double getPerimeter() {
48     return 2 * (width + height);
49   }
50 }
```

The code in Listing 11.4 creates objects of **Circle** and **Rectangle** and invokes the methods on these objects. The **toString()** method is inherited from the **GeometricObject** class and is invoked from a **Circle** object (line 4) and a **Rectangle** object (line 10).

LISTING 11.4 TestCircleRectangle.java

```
1 public class TestCircleRectangle {
2   public static void main(String[] args) {
3     Circle4 circle = new Circle4(1);                          Circle object
4     System.out.println("A circle " + circle.toString());     invoke toString
5     System.out.println("The radius is " + circle.getRadius());
6     System.out.println("The area is " + circle.getArea());
7     System.out.println("The diameter is " + circle.getDiameter());
8
9     Rectangle1 rectangle = new Rectangle1(2, 4);              Rectangle object
10    System.out.println("\nA rectangle " + rectangle.toString());  invoke toString
11    System.out.println("The area is " + rectangle.getArea());
12    System.out.println("The perimeter is " +
13      rectangle.getPerimeter());
14  }
15 }
```

```
A circle created on Thu Sep 24 20:31:02 EDT 2009
color: white and filled: false
The radius is 1.0
The area is 3.141592653589793
The diameter is 2.0

A rectangle created on Thu Sep 24 20:31:02 EDT 2009
color: white and filled: false
The area is 8.0
The perimeter is 12.0
```



The following points regarding inheritance are worthwhile to note:

- Contrary to the conventional interpretation, a subclass is not a subset of its superclass. In fact, a subclass usually contains more information and methods than its superclass. — more in subclass

- Private data fields in a superclass are not accessible outside the class. Therefore, they cannot be used directly in a subclass. They can, however, be accessed/mutated through public accessor/mutator if defined in the superclass. — private data fields

- Not all *is-a* relationships should be modeled using inheritance. For example, a square is a rectangle, but you should not define a `Square` class to extend a `Rectangle` class, because there is nothing to extend (or supplement) from a rectangle to a square. Rather you should define a `Square` class to extend the `GeometricObject` class. For class `A` to extend class `B`, `A` should contain more detailed information than `B`. — nonextensible is-a

- Inheritance is used to model the *is-a* relationship. Do not blindly extend a class just for the sake of reusing methods. For example, it makes no sense for a `Tree` class to extend a `Person` class, even though they share common properties such as height and weight. A subclass and its superclass must have the *is-a* relationship. — no blind extension

- Some programming languages allow you to derive a subclass from several classes. This capability is known as *multiple inheritance*. Java, however, does not allow multiple inheritance. A Java class may inherit directly from only one superclass. This restriction is known as *single inheritance*. If you use the `extends` keyword to define a subclass, it allows only one parent class. Nevertheless, multiple inheritance can be achieved through interfaces, which will be introduced in §14.4, "Interfaces." — multiple inheritance; single inheritance

11.3 Using the `super` Keyword

A subclass inherits accessible data fields and methods from its superclass. Does it inherit constructors? Can superclass constructors be invoked from subclasses? This section addresses these questions and their ramification.

§10.4, "The `this` Reference," introduced the use of the keyword `this` to reference the calling object. The keyword `super` refers to the superclass of the class in which `super` appears. It can be used in two ways:

- To call a superclass constructor.
- To call a superclass method.

11.3.1 Calling Superclass Constructors

The syntax to call a superclass constructor is:

```
super(), or super(parameters);
```

The statement `super()` invokes the no-arg constructor of its superclass, and the statement `super(arguments)` invokes the superclass constructor that matches the `arguments`. The statement `super()` or `super(arguments)` must appear in the first line of the subclass constructor; this is the only way to explicitly invoke a superclass constructor. For example, the constructor in lines 11–15 in Listing 11.2 can be replaced by the following code:

```
public Circle4(double radius, String color, boolean filled) {
  super(color, filled);
  this.radius = radius;
}
```



Caution

You must use the keyword `super` to call the superclass constructor, and the call must be the first statement in the constructor. Invoking a superclass constructor's name in a subclass causes a syntax error.



Note

A constructor is used to construct an instance of a class. Unlike properties and methods, the constructors of a superclass are not inherited in the subclass. They can only be invoked from the constructors of the subclasses, using the keyword `super`.

11.3.2 Constructor Chaining

A constructor may invoke an overloaded constructor or its superclass constructor. If neither is invoked explicitly, the compiler automatically puts `super()` as the first statement in the constructor. For example,

```
public ClassName() {
  // some statements
}
```

Equivalent

```
public ClassName() {
  super();
  // some statements
}
```

```
public ClassName(double d) {
  // some statements
}
```

Equivalent

```
public ClassName(double d) {
  super();
  // some statements
}
```

In any case, constructing an instance of a class invokes the constructors of all the superclasses along the inheritance chain. When constructing an object of a subclass, the subclass constructor first invokes its superclass constructor before performing its own tasks. If the superclass is derived from another class, the superclass constructor invokes its parent-class constructor before performing its own tasks. This process continues until the last constructor along the inheritance hierarchy is called. This is *constructor chaining*.

constructor chaining

Consider the following code:

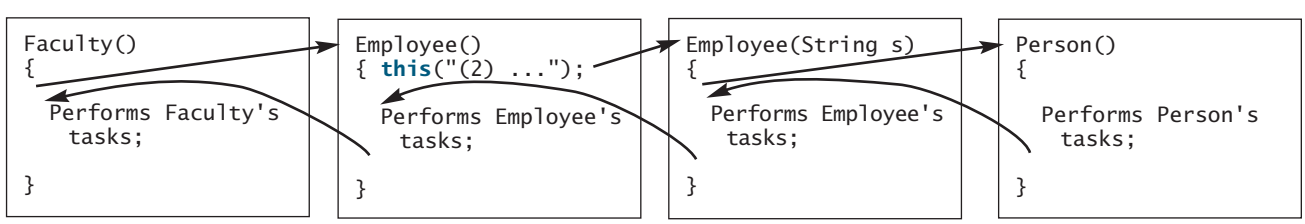
```
1  public class Faculty extends Employee {
2    public static void main(String[] args) {
3      new Faculty();
4    }
5
6    public Faculty() {
7      System.out.println("(4) Performs Faculty's tasks");
8    }
9  }
10
11 class Employee extends Person {
12   public Employee() {
13     this("(2) Invoke Employee's overloaded constructor");
14     System.out.println("(3) Performs Employee's tasks ");
15   }
16
17   public Employee(String s) {
18     System.out.println(s);
19   }
20 }
21
22 class Person {
23   public Person() {
24     System.out.println("(1) Performs Person's tasks");
25   }
26 }
```

invoke overloaded constructor

```
(1) Performs Person's tasks
(2) Invoke Employee's overloaded constructor
(3) Performs Employee's tasks
(4) Performs Faculty's tasks
```



The program produces the preceding output. Why? Let us discuss the reason. In line 3, `new Faculty()` invokes `Faculty`'s no-arg constructor. Since `Faculty` is a subclass of `Employee`, `Employee`'s no-arg constructor is invoked before any statements in `Faculty`'s constructor are executed. `Employee`'s no-arg constructor invokes `Employee`'s second constructor (line 12). Since `Employee` is a subclass of `Person`, `Person`'s no-arg constructor is invoked before any statements in `Employee`'s second constructor are executed. This process is pictured in the figure below.



no-arg constructor



Caution

If a class is designed to be extended, it is better to provide a no-arg constructor to avoid programming errors. Consider the following code:

```
1 public class Apple extends Fruit {
2 }
3
4 class Fruit {
5   public Fruit(String name) {
6     System.out.println("Fruit's constructor is invoked");
7   }
8 }
```

Since no constructor is explicitly defined in `Apple`, `Apple`'s default no-arg constructor is defined implicitly. Since `Apple` is a subclass of `Fruit`, `Apple`'s default constructor automatically invokes `Fruit`'s no-arg constructor. However, `Fruit` does not have a no-arg constructor, because `Fruit` has an explicit constructor defined. Therefore, the program cannot be compiled.

no-arg constructor



Design Guide

It is better to provide a no-arg constructor (if desirable) for every class to make the class easy to extend and to avoid errors.

11.3.3 Calling Superclass Methods

The keyword `super` can also be used to reference a method other than the constructor in the superclass. The syntax is like this:

```
super.method(parameters);
```

You could rewrite the `printCircle()` method in the `Circle` class as follows:

```
public void printCircle() {
  System.out.println("The circle is created " +
    super.getDateCreated() + " and the radius is " + radius);
}
```

It is not necessary to put `super` before `getDateCreated()` in this case, however, because `getDateCreated` is a method in the `GeometricObject` class and is inherited by the `Circle` class. Nevertheless, in some cases, as shown in the next section, the keyword `super` is needed.

11.4 Overriding Methods

A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as *method overriding*.

method overriding

The `toString` method in the `GeometricObject` class returns the string representation for a geometric object. This method can be overridden to return the string representation for a circle. To *override* it, add the following new method in Listing 11.2, Circle4.java:

```
1 public class Circle4 extends GeometricObject1 {
2   // Other methods are omitted
3
4   /** Override the toString method defined in GeometricObject */
5   public String toString() {
6     return super.toString() + "\nradius is " + radius;
7   }
8 }
```

`toString` in superclass

The `toString()` method is defined in the `GeometricObject` class and modified in the `Circle` class. Both methods can be used in the `Circle` class. To invoke the `toString` method defined in the `GeometricObject` class from the `Circle` class, use `super.toString()` (line 6).

Can a subclass of `Circle` access the `toString` method defined in the `GeometricObject` class using a syntax such as `super.super.toString()`? No. This is a syntax error.

no super.super.methodName()

Several points are worth noting:

override accessible instance method

- An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

cannot override static method

- Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden. The hidden static methods can be invoked using the syntax `SuperClassName.staticMethodName`.

11.5 Overriding vs. Overloading

You have learned about overloading methods in §5.8. Overloading means to define multiple methods with the same name but different signatures. Overriding means to provide a new implementation for a method in the subclass. The method is already defined in the superclass.

To override a method, the method must be defined in the subclass using the same signature and the same return type.

Let us use an example to show the differences between overriding and overloading. In (a) below, the method `p(double i)` in class `A` overrides the same method defined in class `B`. In (b), however, the class `B` has two overloaded methods `p(double i)` and `p(int i)`. The method `p(double i)` is inherited from `B`.

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

(b)

When you run the `Test` class in (a), both `a.p(10)` and `a.p(10.0)` invoke the `p(double i)` method defined in class `A` to display `10.0`. When you run the `Test` class in (b), `a.p(10)` invokes the `p(int i)` method defined in class `B` to display `20`, and `a.p(10.0)` invokes the `p(double i)` method defined in class `A` to display `10.0`.

11.6 The `Object` Class and Its `toString()` Method

Every class in Java is descended from the `java.lang.Object` class. If no inheritance is specified when a class is defined, the superclass of the class is `Object` by default. For example, the following two class definitions are the same:

```
public class ClassName {
  ...
}
```

Equivalent

```
public class ClassName extends Object {
  ...
}
```

Classes such as `String`, `StringBuilder`, `Loan`, and `GeometricObject` are implicitly subclasses of `Object` (as are all the main classes you have seen in this book so far). It is important to be familiar with the methods provided by the `Object` class so that you can use them in your classes. We will introduce the `toString` method in the `Object` class in this section.

`toString()`

The signature of the `toString()` method is

```
public String toString()
```

string representation

Invoking `toString()` on an object returns a string that describes the object. By default, it returns a string consisting of a class name of which the object is an instance, an at sign (@), and the object's memory address in hexadecimal. For example, consider the following code for the `Loan` class defined in Listing 10.2:

```
Loan loan = new Loan();
System.out.println(loan.toString());
```

The code displays something like `Loan@15037e5`. This message is not very helpful or informative. Usually you should override the `toString` method so that it returns a descriptive string representation of the object. For example, the `toString` method in the `Object` class was overridden in the `GeometricObject` class in lines 46-49 in Listing 11.1 as follows:

```
public String toString() {
  return "created on " + dateCreated + "\ncolor: " + color +
    " and filled: " + filled;
}
```



Note

print object

You can also pass an object to invoke `System.out.println(object)` or `System.out.print(object)`. This is equivalent to invoking `System.out.println(object.toString())` or `System.out.print(object.toString())`. So you could replace `System.out.println(loan.toString())` with `System.out.println(loan)`.

11.7 Polymorphism

The three pillars of object-oriented programming are encapsulation, inheritance, and polymorphism. You have already learned the first two. This section introduces polymorphism.

First let us define two useful terms: subtype and supertype. A class defines a type. A type defined by a subclass is called a *subtype* and a type defined by its superclass is called a *supertype*. So, you can say that `Circle` is a subtype of `GeometricObject` and `GeometricObject` is a supertype for `Circle`.

subtype
supertype

The inheritance relationship enables a subclass to inherit features from its superclass with additional new features. A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa. For example, every circle is a geometric object, but not every geometric object is a circle. Therefore, you can always pass an instance of a subclass to a parameter of its superclass type. Consider the code in Listing 11.5.

LISTING 11.5 PolymorphismDemo.java

```
1 public class PolymorphismDemo {
2   /** Main method */
3   public static void main(String[] args) {
4     // Display circle and rectangle properties
5     displayObject(new Circle4(1, "red", false));
6     displayObject(new Rectangle1(1, 1, "black", true));
7   }
8
9   /** Display geometric object properties */                    polymorphic call
10  public static void displayObject(GeometricObject1 object) {   polymorphic call
11    System.out.println("Created on " + object.getDateCreated() +
12      ". Color is " + object.getColor());
13  }
14 }
```

```
Created on Mon Mar 09 19:25:20 EDT 2009. Color is white
Created on Mon Mar 09 19:25:20 EDT 2009. Color is black
```



Method `displayObject` (line 10) takes a parameter of the `GeometricObject` type. You can invoke `displayObject` by passing any instance of `GeometricObject` (e.g., `new Circle4(1, "red", false)` and `new Rectangle1(1, 1, "black", false)` in lines 5–6). An object of a subclass can be used wherever its superclass object is used. This is commonly known as *polymorphism* (from a Greek word meaning "many forms"). In simple terms, polymorphism means that a variable of a supertype can refer to a subtype object.

what is polymorphism?

11.8 Dynamic Binding

A method may be defined in a superclass and overridden in its subclass. For example, the `toString()` method is defined in the `Object` class and overridden in `GeometricObject1`. Consider the following code:

```
Object o = new GeometricObject();
System.out.println(o.toString());
```

Which `toString()` method is invoked by `o`? To answer this question, we first introduce two terms: declared type and actual type. A variable must be declared a type. The type of a variable is called its *declared type*. Here `o`'s declared type is `Object`. A variable of a reference type can hold a `null` value or a reference to an instance of the declared type. The instance may be created using the constructor of the declared type or its subtype. The *actual type* of the variable is the actual class for the object referenced by the variable. Here `o`'s actual type is `GeometricObject`, since `o` references to an object created using `new GeometricObject()`. Which `toString()` method is invoked by `o` is determined by `o`'s actual type. This is known as *dynamic binding*.

declared type

actual type

Dynamic binding works as follows: Suppose an object `o` is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n, as shown in Figure 11.2. That is, C_n is the most general class, and C_1 is the most specific

dynamic binding

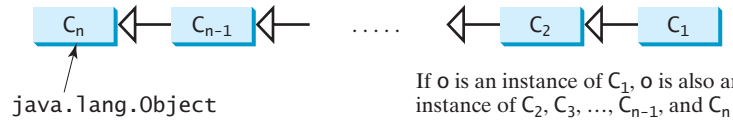


FIGURE 11.2 The method to be invoked is dynamically bound at runtime.

class. In Java, C_n is the `Object` class. If `o` invokes a method `p`, the JVM searches the implementation for the method `p` in C_1, C_2, ..., C_{n-1}, and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.

Listing 11.6 gives an example to demonstrate dynamic binding.



Video Note
polymorphism and dynamic binding demo

LISTING 11.6 DynamicBindingDemo.java

```
1  public class DynamicBindingDemo {
2    public static void main(String[] args) {
3      m(new GraduateStudent());
4      m(new Student());
5      m(new Person());
6      m(new Object());
7    }
8
9    public static void m(Object x) {
10     System.out.println(x.toString());
11   }
12 }
13
14 class GraduateStudent extends Student {
15 }
16
17 class Student extends Person {
18   public String toString() {
19     return "Student";
20   }
21 }
22
23 class Person extends Object {
24   public String toString() {
25     return "Person";
26   }
27 }
```

polymorphic call

dynamic binding

override **toString()**

override **toString()**



```
Student
Student
Person
java.lang.Object@130c19b
```

Method `m` (line 9) takes a parameter of the `Object` type. You can invoke `m` with any object (e.g., `new GraduateStudent()`, `new Student()`, `new Person()`, and `new Object()`) in lines 3–6).

When the method `m(Object x)` is executed, the argument `x`'s `toString` method is invoked. `x` may be an instance of `GraduateStudent`, `Student`, `Person`, or `Object`. Classes `GraduateStudent`, `Student`, `Person`, and `Object` have their own implementations of the `toString` method. Which implementation is used will be determined by `x`'s actual type at runtime. Invoking `m(new GraduateStudent())` (line 3) causes the `toString` method defined in the `Student` class to be invoked.

Invoking `m(new Student())` (line 4) causes the `toString` method defined in the `Student` class to be invoked.

Invoking `m(new Person())` (line 5) causes the `toString` method defined in the `Person` class to be invoked. Invoking `m(new Object())` (line 6) causes the `toString` method defined in the `Object` class to be invoked.

matching vs. binding

Matching a method signature and binding a method implementation are two separate issues. The *declared type* of the reference variable decides which method to match at compile time. The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compile time. A method may be implemented in several subclasses. The JVM dynamically binds the implementation of the method at runtime, decided by the actual type of the variable.

11.9 Casting Objects and the `instanceof` Operator

You have already used the casting operator to convert variables of one primitive type to another. Casting can also be used to convert an object of one class type to another within an inheritance hierarchy. In the preceding section, the statement

```
m(new Student());
```

assigns the object `new Student()` to a parameter of the `Object` type. This statement is equivalent to

```
Object o = new Student(); // Implicit casting
m(o);
```

implicit casting

The statement `Object o = new Student()`, known as *implicit casting*, is legal because an instance of `Student` is automatically an instance of `Object`.

Suppose you want to assign the object reference `o` to a variable of the `Student` type using the following statement:

```
Student b = o;
```

In this case a compile error would occur. Why does the statement `Object o = new Student()` work but `Student b = o` doesn't? The reason is that a `Student` object is always an instance of `Object`, but an `Object` is not necessarily an instance of `Student`. Even though you can see that `o` is really a `Student` object, the compiler is not clever enough to know it. To tell the compiler that `o` is a `Student` object, use an *explicit casting*. The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

explicit casting

```
Student b = (Student)o; // Explicit casting
```

upcasting
downcasting

It is always possible to cast an instance of a subclass to a variable of a superclass (known as *upcasting*), because an instance of a subclass is *always* an instance of its superclass. When casting an instance of a superclass to a variable of its subclass (known as *downcasting*), explicit casting must be used to confirm your intention to the compiler with the (`SubclassName`) cast notation. For the casting to be successful, you must make sure that the object to be cast is an instance of the subclass. If the superclass object is not an instance of the subclass, a runtime `ClassCastException` occurs. For example, if an object is not an instance of `Student`, it cannot be cast into a variable of `Student`. It is a good practice, therefore, to ensure that the object is an instance of another object before attempting a casting. This can be accomplished by using the *instanceof* operator. Consider the following code:

ClassCastException

`instanceof`

```
Object myObject = new Circle();
... // Some lines of code
```

```
/** Perform casting if myObject is an instance of Circle */
if (myObject instanceof Circle) {
  System.out.println("The circle diameter is " +
    ((Circle)myObject).getDiameter());
  ...
}
```

You may be wondering why casting is necessary. Variable `myObject` is declared `Object`. The *declared type* decides which method to match at compile time. Using `myObject.getDiameter()` would cause a compile error, because the `Object` class does not have the `getDiameter` method. The compiler cannot find a match for `myObject.getDiameter()`. It is necessary to cast `myObject` into the `Circle` type to tell the compiler that `myObject` is also an instance of `Circle`.

Why not define `myObject` as a `Circle` type in the first place? To enable generic programming, it is a good practice to define a variable with a supertype, which can accept a value of any subtype.



Note

lowercase keywords

`instanceof` is a Java keyword. Every letter in a Java keyword is in lowercase.



Tip

casting analogy

To help understand casting, you may also consider the analogy of fruit, apple, and orange with the `Fruit` class as the superclass for `Apple` and `Orange`. An apple is a fruit, so you can always safely assign an instance of `Apple` to a variable for `Fruit`. However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of `Fruit` to a variable of `Apple`.

Listing 11.7 demonstrates polymorphism and casting. The program creates two objects (lines 5–6), a circle and a rectangle, and invokes the `displayObject` method to display them (lines 9–10). The `displayObject` method displays the area and diameter if the object is a circle (line 15), and the area if the object is a rectangle (line 21).

LISTING 11.7 CastingDemo.java

```
1  public class CastingDemo {
2    /** Main method */
3    public static void main(String[] args) {
4      // Create and initialize two objects
5      Object object1 = new Circle4(1);
6      Object object2 = new Rectangle1(1, 1);
7
8      // Display circle and rectangle
9      displayObject(object1);
10     displayObject(object2);
11   }
12
13   /** A method for displaying an object */
14   public static void displayObject(Object object) {
15     if (object instanceof Circle4) {
16       System.out.println("The circle area is " +
17         ((Circle4)object).getArea());
18       System.out.println("The circle diameter is " +
19         ((Circle4)object).getDiameter());
20     }
21     else if (object instanceof Rectangle1) {
22       System.out.println("The rectangle area is " +
23         ((Rectangle1)object).getArea());
24     }
25   }
26 }
```

polymorphic call

polymorphic call

```
The circle area is 3.141592653589793
The circle diameter is 2.0
The rectangle area is 1.0
```



The `displayObject(Object object)` method is an example of generic programming. It can be invoked by passing any instance of `Object`.

The program uses implicit casting to assign a `Circle` object to `object1` and a `Rectangle` object to `object2` (lines 5–6), then invokes the `displayObject` method to display the information on these objects (lines 9–10).

In the `displayObject` method (lines 14–25), explicit casting is used to cast the object to `Circle` if the object is an instance of `Circle`, and the methods `getArea` and `getDiameter` are used to display the area and diameter of the circle.

Casting can be done only when the source object is an instance of the target class. The program uses the `instanceof` operator to ensure that the source object is an instance of the target class before performing a casting (line 15).

Explicit casting to `Circle` (lines 17, 19) and to `Rectangle` (line 23) is necessary because the `getArea` and `getDiameter` methods are not available in the `Object` class.



Caution

The object member access operator (`.`) precedes the casting operator. Use parentheses to ensure that casting is done before the `.` operator, as in

. precedes casting

```
((Circle)object).getArea());
```

11.10 The `Object`'s `equals` Method

Another method defined in the `Object` class that is often used is the `equals` method. Its signature is

`equals(Object)`

```
public boolean equals(Object o)
```

This method tests whether two objects are equal. The syntax for invoking it is:

```
object1.equals(object2);
```

The default implementation of the `equals` method in the `Object` class is:

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

This implementation checks whether two reference variables point to the same object using the `==` operator. You should override this method in your custom class to test whether two distinct objects have the same content.

You have already used the `equals` method to compare two strings in §9.2, "The `String` Class." The `equals` method in the `String` class is inherited from the `Object` class and is overridden in the `String` class to test whether two strings are identical in content. You can override the `equals` method in the `Circle` class to compare whether two circles are equal based on their radius as follows:

```
public boolean equals(Object o) {
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;
  }
```

```
    else
      return false;
  }
```

== vs. equals

Note

The `==` comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references. The `equals` method is intended to test whether two objects have the same contents, provided that the method is overridden in the defining class of the objects. The `==` operator is stronger than the `equals` method, in that the `==` operator checks whether the two reference variables refer to the same object.

equals(Object)

Caution

Using the signature `equals(SomeClassName obj)` (e.g., `equals(Circle c)`) to override the `equals` method in a subclass is a common mistake. You should use `equals(Object obj)`. See Review Question 11.15.

11.11 The `ArrayList` Class

Video Note
the `ArrayList` class

Now we are ready to introduce a very useful class for storing objects. You can create an array to store objects. But, once the array is created, its size is fixed. Java provides the `ArrayList` class that can be used to store an unlimited number of objects. Figure 11.3 shows some methods in `ArrayList`.

java.util.ArrayList	
`+ArrayList()`	Creates an empty list.
`+add(o: Object): void`	Appends a new element o at the end of this list.
`+add(index: int, o: Object): void`	Adds a new element o at the specified index in this list.
`+clear(): void`	Removes all the elements from this list.
`+contains(o: Object): boolean`	Returns true if this list contains the element o.
`+get(index: int): Object`	Returns the element from this list at the specified index.
`+indexOf(o: Object): int`	Returns the index of the first matching element in this list.
`+isEmpty(): boolean`	Returns true if this list contains no elements.
`+lastIndexOf(o: Object): int`	Returns the index of the last matching element in this list.
`+remove(o: Object): boolean`	Removes the element o from this list.
`+size(): int`	Returns the number of elements in this list.
`+remove(index: int): boolean`	Removes the element at the specified index.
`+set(index: int, o: Object): Object`	Sets the element at the specified index.

FIGURE 11.3 An `ArrayList` stores an unlimited number of objects.

Listing 11.8 gives an example of using `ArrayList` to store objects.

LISTING 11.8 TestArrayList.java

create `ArrayList`

add element

```
1 public class TestArrayList {
2   public static void main(String[] args) {
3     // Create a list to store cities
4     java.util.ArrayList cityList = new java.util.ArrayList();
5
6     // Add some cities in the list
7     cityList.add("London");
```

```
 8     // cityList now contains [London]
 9     cityList.add("Denver");
10     // cityList now contains [London, Denver]
11     cityList.add("Paris");
12     // cityList now contains [London, Denver, Paris]
13     cityList.add("Miami");
14     // cityList now contains [London, Denver, Paris, Miami]
15     cityList.add("Seoul");
16     // contains [London, Denver, Paris, Miami, Seoul]
17     cityList.add("Tokyo");
18     // contains [London, Denver, Paris, Miami, Seoul, Tokyo]
19
20     System.out.println("List size? " + cityList.size() );                  list size
21     System.out.println("Is Miami in the list? " +
22       cityList.contains("Miami") );                                       contains element?
23     System.out.println("The location of Denver in the list? "
24       + cityList.indexOf("Denver"));                                      element index
25     System.out.println("Is the list empty? " +
26       cityList.isEmpty() ); // Print false                                is empty?
27
28     // Insert a new city at index 2
29     cityList.add(2, "Xian");
30     // contains [London, Denver, Xian, Paris, Miami, Seoul, Tokyo]
31
32     // Remove a city from the list
33     cityList.remove("Miami");                                             remove element
34     // contains [London, Denver, Xian, Paris, Seoul, Tokyo]
35
36     // Remove a city at index 1
37     cityList.remove(1);                                                   remove element
38     // contains [London, Xian, Paris, Seoul, Tokyo]
39
40     // Display the contents in the list
41     System.out.println(cityList.toString());                              toString()
42
43     // Display the contents in the list in reverse order
44     for (int i = cityList.size() - 1; i >= 0; i--)
45       System.out.print(cityList.get(i) + " ");                            get element
46     System.out.println();
47
48     // Create a list to store two circles
49     java.util.ArrayList list = new java.util.ArrayList();                 create ArrayList
50
51     // Add two circles
52     list.add(new Circle4(2));
53     list.add(new Circle4(3));
54
55     // Display the area of the first circle in the list
56     System.out.println("The area of the circle? " +
57       ((Circle4)list.get(0)).getArea());
58   }
59 }
```

```
List size? 6
Is Miami in the list? true
The location of Denver in the list? 1
Is the list empty? false
[London, Xian, Paris, Seoul, Tokyo]
Tokyo Seoul Paris Xian London
The area of the circle? 12.566370614359172
```



The program creates an `ArrayList` using its no-arg constructor (line 4). The `add` method adds any instance of `Object` into the list. Since `String` is a subclass of `Object`, strings can be added to the list. The `add` method (lines 7–17) adds an object to the end of list. So, after `cityList.add("London")` (line 7), the list contains

```
[London]
```

`add(Object)`

After `cityList. add("Denver")` (line 9), the list contains

```
[London, Denver]
```

After adding Paris, Miami, Seoul, and Tokyo (lines 11–17), the list would contain

```
[London, Denver, Paris, Miami, Seoul, Tokyo]
```

`size()`

Invoking `size()` (line 20) returns the size of the list, which is currently `6`. Invoking `contains("Miami")` (line 22) checks whether the object is in the list. In this case, it returns `true`, since `Miami` is in the list. Invoking `indexOf("Denver")` (line 24) returns the index of the object in the list, which is `1`. If the object is not in the list, it returns `-1`. The `isEmpty()` method (line 26) checks whether the list is empty. It returns `false`, since the list is not empty.

add(index, Object)

The statement `cityList.add(2, "Xian")` (line 29) inserts an object to the list at the specified index. After this statement, the list becomes

```
[London, Denver, Xian, Paris, Miami, Seoul, Tokyo]
```

`remove(Object)`

The statement `cityList. remove("Miami")` (line 33) removes the object from the list. After this statement, the list becomes

```
[London, Denver, Xian, Paris, Seoul, Tokyo]
```

`remove(index)`

The statement `cityList. remove(1)` (line 37) removes the object at the specified index from the list. After this statement, the list becomes

```
[London, Xian, Paris, Seoul, Tokyo]
```

The statement in line 41 is same as

```
System.out.println(cityList);
```

`toString()`

The `toString()` method returns a string representation for the list in the form of `[e0.toString(), e1.toString(), ..., ek.toString()]`, where `e0`, `e1`, . . . , and `ek` are the elements in the list.

`getIndex()`

The `get(index)` method (line 45) returns the object at the specified index.



Note

compiler warning

You will get the following warning when compiling this program from the command prompt:

```
Note: TestArrayList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

This warning can be eliminated using generic types discussed in Chapter 21, "Generics". For now, ignore it. Despite the warning, the program will be compiled just fine to produce a .class file.

array vs. `ArrayList`

`ArrayList` objects can be used like arrays, but there are many differences. Table 11.1 lists their similarities and differences.

Once an array is created, its size is fixed. You can access an array element using the square-bracket notation (e.g., `a[index]`). When an `ArrayList` is created, its size is `0`. You cannot use the `get` and `set` methods if the element is not in the list. It is easy to add, insert,

TABLE 11.1 Differences and Similarities between Arrays and `ArrayList`

Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`Object[] a = new Object[10]`	`ArrayList list = new ArrayList();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

and remove elements in a list, but it is rather complex to add, insert, and remove elements in an array. You have to write code to manipulate the array in order to perform these operations.



Note

`java.util.Vector` is also a class for storing objects, which is very similar to the `ArrayList` class. All the methods in `ArrayList` are also available in `Vector`. The `Vector` class was introduced in JDK 1.1. The `ArrayList` class introduced in JDK 1.2 was intended to replace the `Vector` class.

`Vector` class

11.12 A Custom Stack Class



Video Note
the `MyStack` class

"Designing a Class for Stacks" in §10.8 presented a stack class for storing `int` values. This section introduces a stack class to store objects. You can use an `ArrayList` to implement `Stack`, as shown in Listing 11.9. The UML diagram for the class is shown in Figure 11.4.

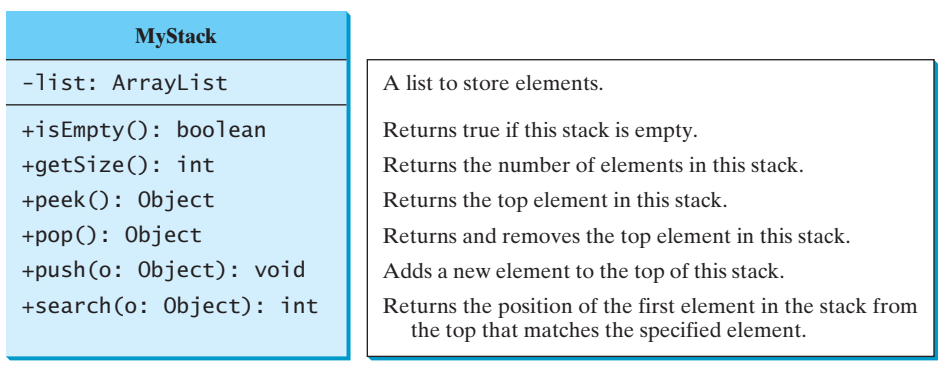

MyStack
-list: ArrayList
+isEmpty(): boolean
+getSize(): int
+peek(): Object
+pop(): Object
+push(o: Object): void
+search(o: Object): int
A list to store elements.
Returns true if this stack is empty.
Returns the number of elements in this stack.
Returns the top element in this stack.
Returns and removes the top element in this stack.
Adds a new element to the top of this stack.
Returns the position of the first element in the stack from the top that matches the specified element.

FIGURE 11.4 The `MyStack` class encapsulates the stack storage and provides the operations for manipulating the stack.

LISTING 11.9 MyStack.java

```
1 public class MyStack {
2   private java.util.ArrayList list = new java.util.ArrayList();
3
4   public boolean isEmpty() {
5     return list.isEmpty();
6   }
```

array list

stack empty?

get stack size

peek stack

remove

push

search

```
 7
 8   public int getSize() {
 9     return list.size();
10   }
11
12   public Object peek() {
13     return list.get(getSize() - 1);
14   }
15
16   public Object pop() {
17     Object o = list.get(getSize() - 1);
18     list.remove(getSize() - 1);
19     return o;
20   }
21
22   public void push(Object o) {
23     list.add(o);
24   }
25
26   public int search(Object o) {
27     return list.lastIndexOf(o);
28   }
29
30   /** Override the toString in the Object class */
31   public String toString() {
32     return "stack: " + list.toString();
33   }
34 }
```

An array list is created to store the elements in the stack (line 2). The `isEmpty()` method (lines 4–6) returns `list.isEmpty()`. The `getSize()` method (lines 8–10) returns `list.size()`. The `peek()` method (lines 12–14) retrieves the element at the top of the stack without removing it. The end of the list is the top of the stack. The `pop()` method (lines 16–20) removes the top element from the stack and returns it. The `push(Object element)` method (lines 22–24) adds the specified element to the stack. The `search(Object element)` method checks whether the specified element is in the stack, and it returns the index of first-matching element in the stack from the top by invoking `list.lastIndexOf(o)`. The `toString()` method (lines 31–33) defined in the `Object` class is overridden to display the contents of the stack by invoking `list.toString()`. The `toString()` method implemented in `ArrayList` returns a string representation of all the elements in an array list.



Design Guide

In Listing 11.9, `MyStack` contains `ArrayList`. The relationship between `MyStack` and `ArrayList` is *composition*. While inheritance models an is-a relationship, composition models a has-a relationship. You may also implement `MyStack` as a subclass of `ArrayList` (see Exercise 11.4). Using composition is better, however, because it enables you to define a completely new stack class without inheriting the unnecessary and inappropriate methods from `ArrayList`.

composition
has-a
is-a

11.13 The `protected` Data and Methods

So far you have used the `private` and `public` keywords to specify whether data fields and methods can be accessed from the outside of the class. Private members can be accessed only from the inside of the class, and public members can be accessed from any other classes.

Often it is desirable to allow subclasses to access data fields or methods defined in the superclass, but not allow nonsubclasses to access these data fields and methods. To do so, you can use the `protected` keyword. A protected data field or method in a superclass can be accessed in its subclasses.

why **`protected`**?

The modifiers **private**, **protected**, and **public** are known as *visibility* or *accessibility modifiers* because they specify how class and class members are accessed. The visibility of these modifiers increases in this order:

Visibility increases →

private, none (if no modifier is used), protected, public

Table 11.2 summarizes the accessibility of the members in a class. Figure 11.5 illustrates how a public, protected, default, and private datum or method in class **C1** can be accessed from a class **C2** in the same package, from a subclass **C3** in the same package, from a subclass **C4** in a different package, and from a class **C5** in a different package.

TABLE 11.2 Data and Methods Visibility

Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
(default)	✓	✓	–	–
private	✓	–	–	–

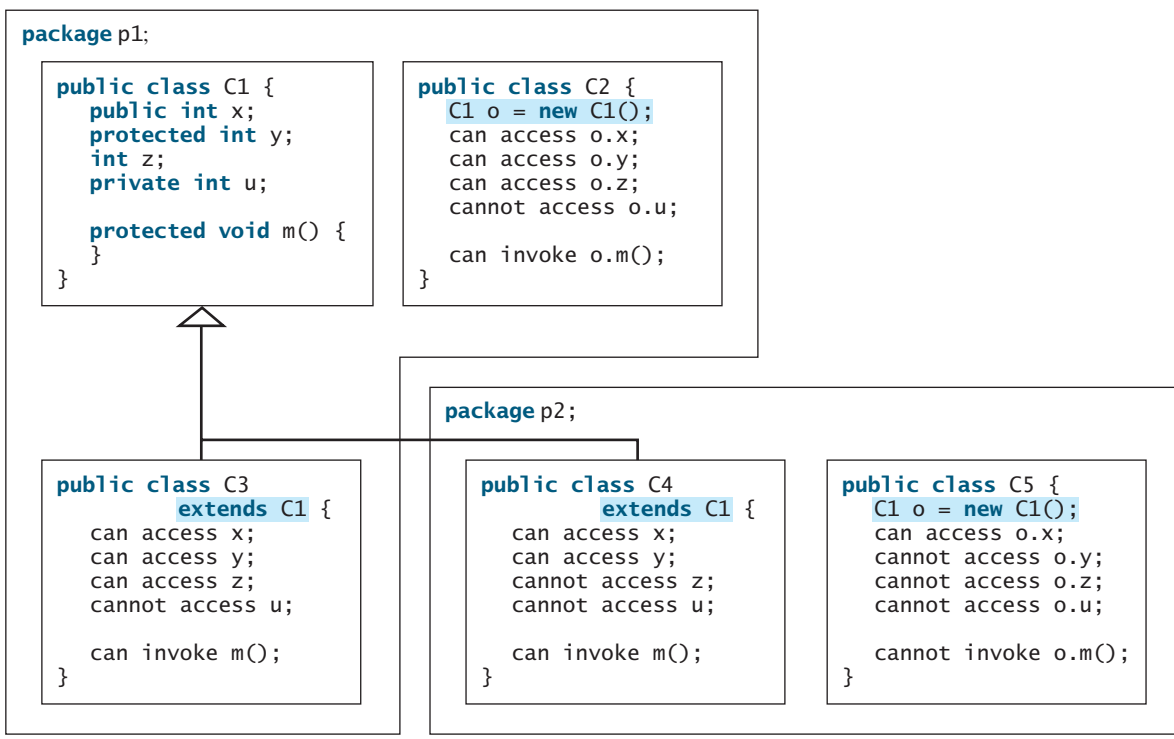


FIGURE 11.5 Visibility modifiers are used to control how data and methods are accessed.

Use the **private** modifier to hide the members of the class completely so that they cannot be accessed directly from outside the class. Use no modifiers in order to allow the members of the class to be accessed directly from any class within the same package but not from other

packages. Use the `protected` modifier to enable the members of the class to be accessed by the subclasses in any package or classes in the same package. Use the `public` modifier to enable the members of the class to be accessed by any class.

Your class can be used in two ways: for creating instances of the class, and for defining subclasses by extending the class. Make the members `private` if they are not intended for use from outside the class. Make the members `public` if they are intended for the users of the class. Make the fields or methods `protected` if they are intended for the extenders of the class but not the users of the class.

The `private` and `protected` modifiers can be used only for members of the class. The `public` modifier and the default modifier (i.e., no modifier) can be used on members of the class as well on the class. A class with no modifier (i.e., not a public class) is not accessible by classes from other packages.



Note

change visibility

A subclass may override a protected method in its superclass and change its visibility to public. However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

11.14 Preventing Extending and Overriding

You may occasionally want to prevent classes from being extended. In such cases, use the `final` modifier to indicate that a class is final and cannot be a parent class. The `Math` class is a final class. The `String`, `StringBuilder`, and `StringBuffer` classes are also final classes. For example, the following class is final and cannot be extended:

```
public final class C {
  // Data fields, constructors, and methods omitted
}
```

You also can define a method to be final; a final method cannot be overridden by its subclasses.

For example, the following method is final and cannot be overridden:

```
public class Test {
  // Data fields, constructors, and methods omitted

  public final void m() {
    // Do something
  }
}
```



Note

The modifiers are used on classes and class members (data and methods), except that the `final` modifier can also be used on local variables in a method. A final local variable is a constant inside a method.

KEY TERMS

actual type 385
array list 393
casting objects 387
composition 394
constructor chaining 381
declared type 385
dynamic binding 385
`final` 396

has-a relationship 394
inheritance 374
`instanceof` 387
is-a relationship 394
override 382
polymorphism 385
`protected` 394
subclass 375
subtype 384
superclass 375
supertype 384
vector 393

Chapter Summary

1. You can derive a new class from an existing class. This is known as class inheritance. The new class is called a subclass, child class or extended class. The existing class is called a *superclass*, *parent class*, or *base class*.

2. A constructor is used to construct an instance of a class. Unlike properties and methods, the constructors of a superclass are not inherited in the subclass. They can be invoked only from the constructors of the subclasses, using the keyword `super`.

3. A constructor may invoke an overloaded constructor or its superclass's constructor. The call must be the first statement in the constructor. If none of them is invoked explicitly, the compiler puts `super()` as the first statement in the constructor, which invokes the superclass's no-arg constructor.

4. To override a method, the method must be defined in the subclass using the same signature as in its superclass.

5. An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

6. Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.

7. Every class in Java is descended from the `java.lang.Object` class. If no inheritance is specified when a class is defined, its superclass is `Object`.

8. If a method's parameter type is a superclass (e.g., `Object`), you may pass an object to this method of any of the parameter's subclasses (e.g., `Circle` or `String`). When an object (e.g., a `Circle` object or a `String` object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., `toString`) is determined dynamically.

9. It is always possible to cast an instance of a subclass to a variable of a superclass, because an instance of a subclass is *always* an instance of its superclass. When casting an instance of a superclass to a variable of its subclass, explicit casting must be used to confirm your intention to the compiler with the (`SubclassName`) cast notation.

10. A class defines a type. A type defined by a subclass is called a *subtype* and a type defined by its superclass is called a *supertype*.

11. When invoking an instance method from a reference variable, the *actual type of* the variable decides which implementation of the method is used *at runtime*. When

accessing a field or a static method, the *declared type* of the reference variable decides which method is used *at compile time*.

12. You can use `obj instanceof AClass` to test whether an object is an instance of a class.

13. You can use the `protected` modifier to prevent the data and methods from being accessed by nonsubclasses from a different package.

14. You can use the `final` modifier to indicate that a class is final and cannot be a parent class and to indicate that a method is final and cannot be overridden.

Review Questions

Sections 11.2–11.5

11.1 What is the printout of running the class `C` in (a)? What problem arises in compiling the program in (b)?

```
class A {
  public A() {
    System.out.println(
      "A's no-arg constructor is invoked");
  }
}

class B extends A {
}

public class C {
  public static void main(String[] args) {
    B b = new B();
  }
}
```

(a)

```
class A {
  public A(int x) {
  }
}

class B extends A {
  public B() {
  }
}

public class C {
  public static void main(String[] args) {
    B b = new B();
  }
}
```

(b)

11.2 True or false?

1. A subclass is a subset of a superclass.
2. When invoking a constructor from a subclass, its superclass's no-arg constructor is always invoked.
3. You can override a private method defined in a superclass.
4. You can override a static method defined in a superclass.

11.3 Identify the problems in the following classes:

```
 1 public class Circle {
 2   private double radius;
 3
 4   public Circle(double radius) {
 5     radius = radius;
 6   }
 7
 8   public double getRadius() {
 9     return radius;
10   }
11
12   public double getArea() {
13     return radius * radius * Math.PI;
14   }
```

```
15 }
16
17 class B extends Circle {
18   private double length;
19
20   B(double radius, double length) {
21     Circle(radius);
22     length = length;
23   }
24
25   /** Override getArea() */
26   public double getArea() {
27     return getArea() * length;
28   }
29 }
```

11.4 How do you explicitly invoke a superclass's constructor from a subclass?

11.5 How do you invoke an overridden superclass method from a subclass?

11.6 Explain the difference between method overloading and method overriding.

11.7 If a method in a subclass has the same signature as a method in its superclass with the same return type, is the method overridden or overloaded?

11.8 If a method in a subclass has the same signature as a method in its superclass with a different return type, will this be a problem?

11.9 If a method in a subclass has the same name as a method in its superclass with different parameter types, is the method overridden or overloaded?

Sections 11.6–11.9

11.10 Does every class have a `toString` method and an `equals` method? Where do they come from? How are they used? Is it appropriate to override these methods?

11.11 Show the output of following program:

```
1 public class Test {
2   public static void main(String[] args) {
3     A a = new A(3);
4   }
5 }
6
7 class A extends B {
8   public A(int t) {
9     System.out.println("A's constructor is invoked");
10  }
11 }
12
13 class B {
14  public B() {
15    System.out.println("B's constructor is invoked");
16  }
17 }
```

Is the no-arg constructor of `Object` invoked when `new A(3)` is invoked?

11.12 For the `GeometricObject` and `Circle` classes in Listings 11.1 and 11.2, answer the following questions:

(a) Are the following Boolean expressions true or false?

```
Circle circle = new Circle(1);
GeometricObject object1 = new GeometricObject();
```

```
(circle instanceof GeometricObject)
(object1 instanceof GeometricObject)
(circle instanceof Circle)
(object1 instanceof Circle)
```

(b) Can the following statements be compiled?

```
Circle circle = new Circle(5);
GeometricObject object = circle;
```

(c) Can the following statements be compiled?

```
GeometricObject object = new GeometricObject();
Circle circle = (Circle)object;
```

11.13 Suppose that `Fruit`, `Apple`, `Orange`, `GoldenDelicious`, and `Macintosh` are declared, as shown in Figure 11.6.

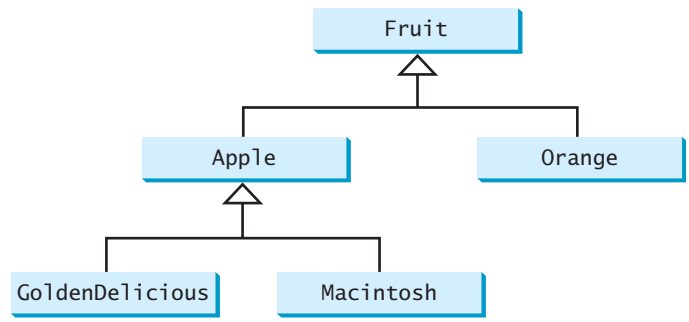


FIGURE 11.6 `GoldenDelicious` and `Macintosh` are subclasses of `Apple`; `Apple` and `Orange` are subclasses of `Fruit`.

Assume that the following declaration is given:

```
Fruit fruit = new GoldenDelicious();
Orange orange = new Orange();
```

Answer the following questions:

(1) Is `fruit instanceof Fruit`?
(2) Is `fruit instanceof Orange`?
(3) Is `fruit instanceof Apple`?
(4) Is `fruit instanceof GoldenDelicious`?
(5) Is `fruit instanceof Macintosh`?
(6) Is `orange instanceof Orange`?
(7) Is `orange instanceof Fruit`?
(8) Is `orange instanceof Apple`?
(9) Suppose the method `makeAppleCider` is defined in the `Apple` class. Can `fruit` invoke this method? Can `orange` invoke this method?
(10) Suppose the method `makeOrangeJuice` is defined in the `Orange` class. Can `orange` invoke this method? Can `fruit` invoke this method?
(11) Is the statement `Orange p = new Apple()` legal?
(12) Is the statement `Macintosh p = new Apple()` legal?
(13) Is the statement `Apple p = new Macintosh()` legal?

11.14 What is wrong in the following code?

```
1 public class Test {
2   public static void main(String[] args) {
3     Object fruit = new Fruit();
4     Object apple = (Apple)fruit;
5   }
6 }
7
8 class Apple extends Fruit {
9 }
10
11 class Fruit {
12 }
```

Section 11.10

11.15 When overriding the `equals` method, a common mistake is mistyping its signature in the subclass. For example, the `equals` method is incorrectly written as `equals(Circle circle)`, as shown in (a) in the code below; instead, it should be `equals(Object circle)`, as shown in (b). Show the output of running class `Test` with the `Circle` class in (a) and in (b), respectively.

```
public class Test {
  public static void main(String[] args) {
    Object circle1 = new Circle();
    Object circle2 = new Circle();
    System.out.println(circle1.equals(circle2));
  }
}
```

```
class Circle {
  double radius;

  public boolean equals(Circle circle) {
    return this.radius == circle.radius;
  }
}
```

(a)

```
class Circle {
  double radius;

  public boolean equals(Object circle) {
    return this.radius ==
      ((Circle)circle).radius;
  }
}
```

(b)

Sections 11.11–11.12

11.16 How do you create an `ArrayList`? How do you append an object to a list? How do you insert an object at the beginning of a list? How do you find the number of objects in a list? How do you remove a given object from a list? How do you remove the last object from the list? How do you check whether a given object is in a list? How do you retrieve an object at a specified index from a list?

11.17 There are three errors in the code below. Identify them.

```
ArrayList list = new ArrayList();
list.add("Denver");
list.add("Austin");
list.add(new java.util.Date());
String city = list.get(0);
list.set(3, "Dallas");
System.out.println(list.get(3));
```

Sections 11.13–11.14

11.18 What modifier should you use on a class so that a class in the same package can access it, but a class in a different package cannot access it?

11.19 What modifier should you use so that a class in a different package cannot access the class, but its subclasses in any package can access it?

11.20 In the code below, classes `A` and `B` are in the same package. If the question marks are replaced by blanks, can class `B` be compiled? If the question marks are replaced by `private`, can class `B` be compiled? If the question marks are replaced by `protected`, can class `B` be compiled?

```
package p1;

public class A {
  ?   int i;
  ?   void m() {
    ...
  }
}
```

(a)

```
package p1;

public class B extends A {
  public void m1(String[] args) {
    System.out.println(i);
    m();
  }
}
```

(b)

11.21 In the code below, classes `A` and `B` are in different packages. If the question marks are replaced by blanks, can class `B` be compiled? If the question marks are replaced by `private`, can class `B` be compiled? If the question marks are replaced by `protected`, can class `B` be compiled?

```
package p1;

public class A {
  ?   int i;
  ?   void m() {
    ...
  }
}
```

(a)

```
package p2;

public class B extends A {
  public void m1(String[] args) {
    System.out.println(i);
    m();
  }
}
```

(b)

11.22 How do you prevent a class from being extended? How do you prevent a method from being overridden?

Comprehensive

11.23 Define the following terms: inheritance, superclass, subclass, the keywords `super` and `this`, casting objects, the modifiers `protected` and `final`.

11.24 Indicate true or false for the following statements:

- A protected datum or method can be accessed by any class in the same package.
- A protected datum or method can be accessed by any class in different packages.
- A protected datum or method can be accessed by its subclasses in any package.
- A final class can have instances.
- A final class can be extended.
- A final method can be overridden.
- You can always successfully cast an instance of a subclass to a superclass.
- You can always successfully cast an instance of a superclass to a subclass.

11.25 Describe the difference between method matching and method binding.

11.26 What is polymorphism? What is dynamic binding?

Programming Exercises

Sections 11.2–11.4

11.1 (*The `Triangle` class*) Design a class named `Triangle` that extends `GeometricObject`. The class contains:

- Three `double` data fields named `side1`, `side2`, and `side3` with default values `1.0` to denote three sides of the triangle.
- A no-arg constructor that creates a default triangle.
- A constructor that creates a triangle with the specified `side1`, `side2`, and `side3`.
- The accessor methods for all three data fields.
- A method named `getArea()` that returns the area of this triangle.
- A method named `getPerimeter()` that returns the perimeter of this triangle.
- A method named `toString()` that returns a string description for the triangle.

For the formula to compute the area of a triangle, see Exercise 2.21. The `toString()` method is implemented as follows:

```
return "Triangle: side1 = " + side1 + " side2 = " + side2 +
  " side3 = " + side3;
```

Draw the UML diagram for the classes `Triangle` and `GeometricObject`. Implement the class. Write a test program that creates a `Triangle` object with sides `1`, `1.5`, `1`, color `yellow` and `filled true`, and displays the area, perimeter, color, and whether filled or not.

Sections 11.5–11.11

11.2 (*The `Person`, `Student`, `Employee`, `Faculty`, and `Staff` classes*) Design a class named `Person` and its two subclasses named `Student` and `Employee`. Make `Faculty` and `Staff` subclasses of `Employee`. A person has a name, address, phone number, and email address. A student has a class status (freshman, sophomore, junior, or senior). Define the status as a constant. An employee has an office, salary, and date hired. Define a class named `MyDate` that contains the fields `year`, `month`, and `day`. A faculty member has office hours and a rank. A staff member has a title. Override the `toString` method in each class to display the class name and the person's name.

Draw the UML diagram for the classes. Implement the classes. Write a test program that creates a `Person`, `Student`, `Employee`, `Faculty`, and `Staff`, and invokes their `toString()` methods.

11.3 (*Subclasses of `Account`*) In Exercise 8.7, the `Account` class was defined to model a bank account. An account has the properties account number, balance, annual interest rate, and date created, and methods to deposit and withdraw funds. Create two subclasses for checking and saving accounts. A checking account has an overdraft limit, but a savings account cannot be overdrawn.

Draw the UML diagram for the classes. Implement the classes. Write a test program that creates objects of `Account`, `SavingsAccount`, and `CheckingAccount` and invokes their `toString()` methods.

11.4 (*Implementing `MyStack` using inheritance*) In Listing 11.9, `MyStack` is implemented using composition. Create a new stack class that extends `ArrayList`.

Draw the UML diagram for the classes. Implement `MyStack`. Write a test program that prompts the user to enter five strings and displays them in reverse order.

11.5 (*The `Course` class*) Rewrite the `Course` class in Listing 10.6. Use an `ArrayList` to replace an array to store students. You should not change the original contract of the `Course` class (i.e., the definition of the constructors and methods should not be changed).

11.6 (*Using `ArrayList`*) Write a program that creates an `ArrayList` and adds a `Loan` object, a `Date` object, a string, a `JFrame` object, and a `Circle` object to the list, and use a loop to display all the elements in the list by invoking the object's `toString()` method.

11.7* (*Implementing `ArrayList`*) `ArrayList` is implemented in the Java API. Implement `ArrayList` and the methods defined in Figure 11.3. (*Hint*: Use an array to store the elements in `ArrayList`. If the size of the `ArrayList` exceeds the capacity of the current array, create a new array that doubles the size of the current array and copy the contents of the current to the new array.)

11.8** (*New `Account` class*) An `Account` class was specified in Exercise 8.7. Design a new `Account` class as follows:

- Add a new data field `name` of the `String` type to store the name of the customer.
- Add a new constructor that constructs an account with the specified name, id, and balance.
- Add a new data field named `transactions` whose type is `ArrayList` that stores the transaction for the accounts. Each transaction is an instance of the `Transaction` class. The `Transaction` class is defined as shown in Figure 11.7.
- Modify the `withdraw` and `deposit` methods to add a transaction to the `transactions` array list.
- All other properties and methods are same as in Exercise 8.7.

Write a test program that creates an `Account` with annual interest rate `1.5%`, balance `1000`, id `1122`, and name `George`. Deposit $30, $40, $50 to the account and withdraw $5, $4, $2 from the account. Print an account summary that shows account holder name, interest rate, balance, and all transactions.

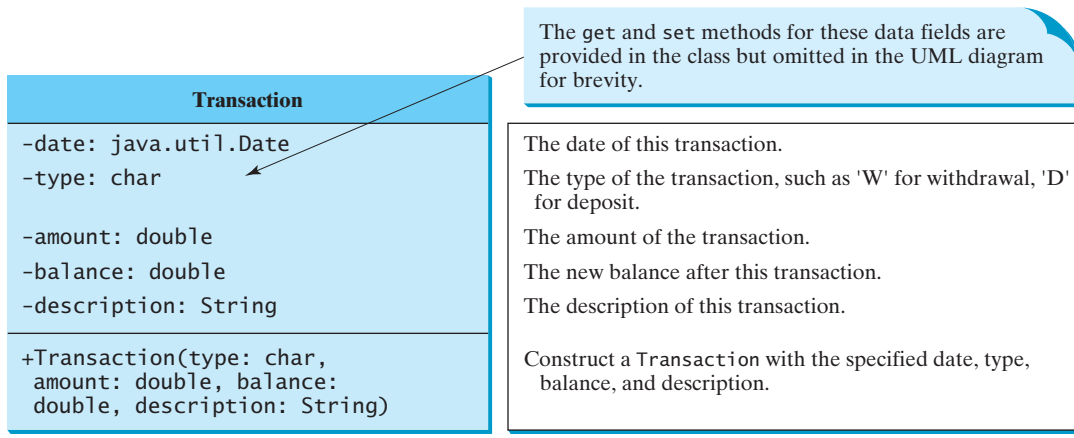

The get and set methods for these data fields are provided in the class but omitted in the UML diagram for brevity.
Transaction
-date: java.util.Date
-type: char
-amount: double
-balance: double
-description: String
+Transaction(type: char, amount: double, balance: double, description: String)
The date of this transaction.
The type of the transaction, such as 'W' for withdrawal, 'D' for deposit.
The amount of the transaction.
The new balance after this transaction.
The description of this transaction.
Construct a Transaction with the specified date, type, balance, and description.

FIGURE 11.7 The `Transaction` class describes a transaction for a bank account.

CHAPTER 12

GUI BASICS

Objectives

- To distinguish between Swing and AWT (§12.2).
- To describe the Java GUI API hierarchy (§12.3).
- To create user interfaces using frames, panels, and simple GUI components (§12.4).
- To understand the role of layout managers (§12.5).
- To use the `FlowLayout`, `GridLayout`, and `BorderLayout` managers to lay out components in a container (§12.5).
- To use `JPanel` to make panels as subcontainers (§12.6).
- To specify colors and fonts using the `Color` and `Font` classes (§§12.7–12.8).
- To apply common features such as borders, tool tips, fonts, and colors on Swing components (§12.9).
- To use borders to visually group user-interface components (§12.9).
- To create image icons using the `ImageIcon` class (§12.10).



12.1 Introduction

The design of the API for Java GUI programming is an excellent example of how the object-oriented principle is applied. This chapter serves two purposes. First, it introduces the basics of Java GUI programming. Second, it uses GUI to demonstrate OOP. Specifically, this chapter will introduce the framework of Java GUI API and discuss GUI components and their relationships, containers and layout managers, colors, fonts, borders, image icons, and tool tips.

12.2 Swing vs. AWT

We used simple GUI examples to demonstrate OOP in §8.6.3, "Displaying GUI Components." We used the GUI components such as `JButton`, `JLabel`, `JTextField`, `JRadioButton`, and `JComboBox`. Why do the GUI component classes have the prefix *J*? Instead of `JButton`, why not name it simply `Button`? In fact, there is a class already named `Button` in the `java.awt` package.

When Java was introduced, the GUI classes were bundled in a library known as the Abstract Windows Toolkit (AWT). AWT is fine for developing simple graphical user interfaces, but not for developing comprehensive GUI projects. Besides, AWT is prone to platform-specific bugs. The AWT user-interface components were replaced by a more robust, versatile, and flexible library known as *Swing components*. Swing components are painted directly on canvases using Java code, except for components that are subclasses of `java.awt.Window` or `java.awt.Panel`, which must be drawn using native GUI on a specific platform. Swing components depend less on the target platform and use less of the native GUI resource. For this reason, Swing components that don't rely on native GUI are referred to as *lightweight components,* and AWT components are referred to as *heavyweight components*.

Swing components

lightweight
heavyweight

To distinguish new Swing component classes from their AWT counterparts, the Swing GUI component classes are named with a prefixed *J*. Although AWT components are still supported in Java, it is better to learn to how program using Swing components, because the AWT user-interface components will eventually fade away. This book uses Swing GUI components exclusively.

why prefix J?

12.3 The Java GUI API

The GUI API contains classes that can be classified into three groups: *component classes, container classes*, and *helper classes*. Their hierarchical relationships are shown in Figure 12.1.

The component classes, such as `JButton`, `JLabel`, and `JTextField`, are for creating the user interface. The container classes, such as `JFrame`, `JPanel`, and `JApplet`, are used to contain other components. The helper classes, such as `Graphics`, `Color`, `Font`, `FontMetrics`, and `Dimension`, are used to support GUI components.



Note

The `JFrame`, `JApplet`, `JDialog`, and `JComponent` classes and their subclasses are grouped in the `javax.swing` package. All the other classes in Figure 12.1 are grouped in the `java.awt` package.

12.3.1 Component Classes

An instance of `Component` can be displayed on the screen. `Component` is the root class of all the user-interface classes including container classes, and `JComponent` is the root class of all the lightweight Swing components. Both `Component` and `JComponent` are abstract classes. Abstract classes will be introduced in Chapter 14, "Abstract Classes and Interfaces." For now, all you need to know is that abstract classes are same as classes except that you cannot create instances using the `new` operator. For example, you cannot use `new JComponent()` to create an

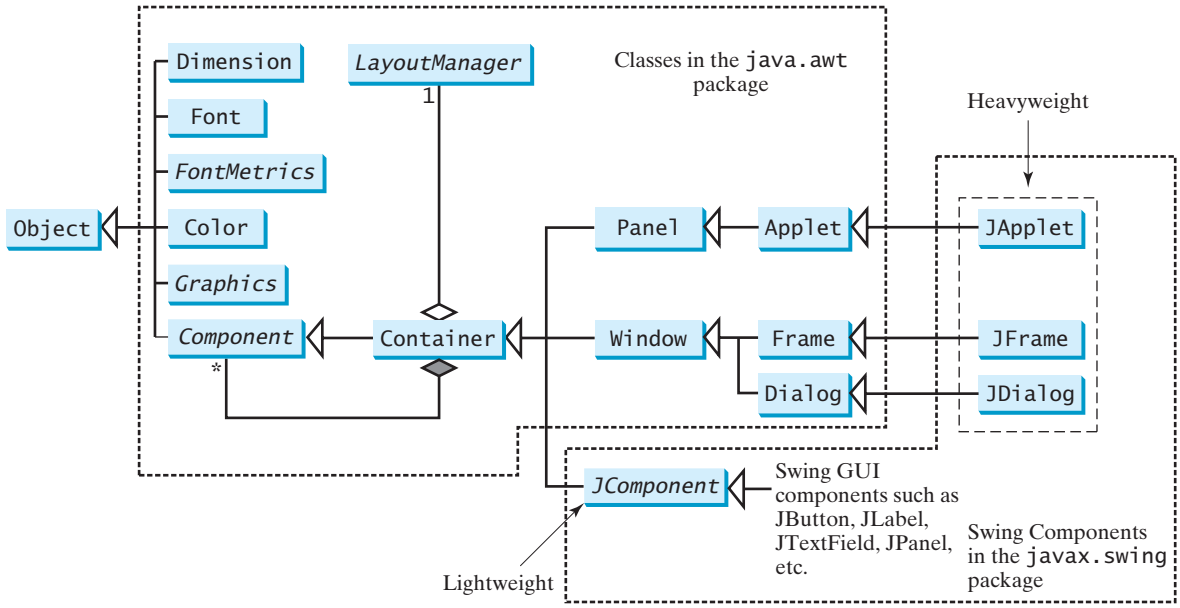


FIGURE 12.1 Java GUI programming utilizes the classes shown in this hierarchical diagram.

instance of `JComponent`. However, you can use the constructors of concrete subclasses of `JComponent` to create `JComponent` instances. It is important to become familiar with the class inheritance hierarchy. For example, the following statements all display true:

```
JButton jbtOK = new JButton("OK");
System.out.println(jbtOK instanceof JButton);
System.out.println(jbtOK instanceof JComponent);
System.out.println(jbtOK instanceof Container);
System.out.println(jbtOK instanceof Component);
System.out.println(jbtOK instanceof Object);
```

12.3.2 Container Classes

An instance of `Container` can hold instances of `Component`. Container classes are GUI components that are used to contain other GUI components. `Window`, `Panel`, `Applet`, `Frame`, and `Dialog` are the container classes for AWT components. To work with Swing components, use `Container`, `JFrame`, `JDialog`, `JApplet`, and `JPanel`, as described in Table 12.1.

TABLE 12.1 GUI Container Classes

Container Class	*Description*
`java.awt.Container`	is used to group components. Frames, panels, and applets are its subclasses.
`javax.swing.JFrame`	is a window not contained inside another window. It is used to hold other Swing user-interface components in Java GUI applications.
`javax.swing.JPanel`	is an invisible container that holds user-interface components. Panels can be nested. You can place panels inside a container that includes a panel. `JPanel` is also often used as a canvas to draw graphics.
`javax.swing.JApplet`	is a subclass of `Applet`. You must extend `JApplet` to create a Swing-based Java applet.
`javax.swing.JDialog`	is a popup window or message box generally used as a temporary window to receive additional information from the user or to provide notification that an event has occurred.

12.3.3 GUI Helper Classes

The helper classes, such as `Graphics`, `Color`, `Font`, `FontMetrics`, `Dimension`, and `LayoutManager`, are not subclasses of `Component`. They are used to describe the properties of GUI components, such as graphics context, colors, fonts, and dimension, as described in Table 12.2.

TABLE 12.2 GUI Helper Classes

Helper Class	*Description*
`java.awt.Graphics`	is an abstract class that provides the methods for drawing strings, lines, and simple shapes.
`java.awt.Color`	deals with the colors of GUI components. For example, you can specify background or foreground colors in components like `JFrame` and `JPanel`, or you can specify colors of lines, shapes, and strings in drawings.
`java.awt.Font`	specifies fonts for the text and drawings on GUI components. For example, you can specify the font type (e.g., SansSerif), style (e.g., bold), and size (e.g., 24 points) for the text on a button.
`java.awt.FontMetrics`	is an abstract class used to get the properties of the fonts.
`java.awt.Dimension`	encapsulates the width and height of a component (in integer precision) in a single object.
`java.awt.LayoutManager`	specifies how components are arranged in a container.



Note

The helper classes are in the `java.awt` package. The Swing components do not replace all the classes in AWT, only the AWT GUI component classes (e.g., `Button`, `TextField`, `TextArea`). The AWT helper classes are still useful in GUI programming.

12.4 Frames

To create a user interface, you need to create either a frame or an applet to hold the user-interface components. Creating Java applets will be introduced in Chapter 18, "Applets and Multimedia." This section introduces frames.

12.4.1 Creating a Frame

To create a frame, use the `JFrame` class, as shown in Figure 12.2.

The program in Listing 12.1 creates a frame:

LISTING 12.1 MyFrame.java

import package

```
1 import javax.swing.JFrame;
2
3 public class MyFrame {
4   public static void main(String[] args) {
```

create frame

```
5     JFrame frame = new JFrame("MyFrame"); // Create a frame
```

set size

```
6     frame.setSize(400, 300); // Set the frame size
```

center frame

```
7     frame.setLocationRelativeTo(null); // Center a frame
```

close upon exit

```
8     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

display the frame

```
9     frame.setVisible(true); // Display the frame
10  }
11 }
```

javax.swing.JFrame	
+JFrame()	Creates a default frame with no title.
+JFrame(title: String)	Creates a frame with the specified title.
+setSize(width: int, height: int): void	Sets the size of the frame.
+setLocation(x: int, y: int): void	Sets the upper-left-corner location of the frame.
+setVisible(visible: boolean): void	Sets true to display the frame.
+setDefaultCloseOperation(mode: int): void	Specifies the operation when the frame is closed.
+setLocationRelativeTo(c: Component): void	Sets the location of the frame relative to the specified component. If the component is null, the frame is centered on the screen.
+pack(): void	Automatically sets the frame size to hold the components in the frame.

FIGURE 12.2 `JFrame` is a top-level container to hold GUI components.

The frame is not displayed until the `frame.setVisible(true)` method is invoked. `frame.setSize(400, 300)` specifies that the frame is `400` pixels wide and `300` pixels high. If the `setSize` method is not used, the frame will be sized to display just the title bar. Since the `setSize` and `setVisible` methods are both defined in the `Component` class, they are inherited by the `JFrame` class. Later you will see that these methods are also useful in many other subclasses of `Component`.

When you run the `MyFrame` program, a window will be displayed on the screen (see Figure 12.3(a)).



FIGURE 12.3 (a) The program creates and displays a frame with the title `MyFrame`. (b) An OK button is added to the frame.

Invoking `setLocationRelativeTo(null)` (line 7) centers the frame on the screen. Invoking `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)` (line 8) tells the program to terminate when the frame is closed. If this statement is not used, the program does not terminate when the frame is closed. In that case, you have to stop the program by pressing *Ctrl+C* at the DOS prompt window in Windows or stop the process by using the kill command in Unix. If you run the program from an IDE such as Eclipse or NetBeans, you need to click the red Terminate button in the Console pane to stop the program.



Note

Recall that a pixel is the smallest unit of space available for drawing on the screen. You can think of a pixel as a small rectangle and think of the screen as paved with pixels. The *resolution* specifies the number of pixels per square inch. The more pixels the screen has, the higher the screen's resolution. The higher the resolution, the finer the detail you can see.

pixel and resolution



Note

You should invoke the `setSize(w, h)` method before invoking `setLocationRelativeTo(null)` to center the frame.

`setSize` before centering

12.4.2 Adding Components to a Frame

The frame shown in Figure 12.3(a) is empty. Using the **add** method, you can add components into the frame, as in Listing 12.2.

LISTING 12.2 MyFrameWithComponents.java

```
1  import javax.swing.*;
2
3  public class MyFrameWithComponents {
4    public static void main(String[] args) {
5      JFrame frame = new JFrame("MyFrameWithComponents");
6
7      // Add a button into the frame
8      JButton jbtOK = new JButton("OK");
9      frame.add(jbtOK);
10
11     frame.setSize(400, 300);
12     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13     frame.setLocationRelativeTo(null); // Center the frame
14     frame.setVisible(true);
15   }
16 }
```

create a button
add to frame

set size
exit upon closing window
center the frame
set visible

Each **JFrame** contains a content pane. A content pane is an instance of **java.awt.Container**. The GUI components such as buttons are placed in the content pane in a frame. In earlier version of Java, you had to use the **getContentPane** method in the **JFrame** class to return the content pane of the frame, then invoke the content pane's **add** method to place a component into the content pane, as follows:

```
java.awt.Container container = frame.getContentPane();
container.add(jbtOK);
```

This was cumbersome. The new version of Java since Java 5 allows you to place components into the content pane by invoking a frame's **add** method, as follows:

```
frame.add(jbtOK);
```

content-pane delegation

This new feature is called *content-pane delegation*. Strictly speaking, a component is added into the content pane of a frame. For simplicity we say that a component is added to a frame.

An object of **JButton** was created using **new JButton("OK")**, and this object was added to the content pane of the frame (line 9).

The **add(Component comp)** method defined in the **Container** class adds an instance of **Component** to the container. Since **JButton** is a subclass of **Component**, an instance of **JButton** is also an instance of **Component**. To remove a component from a container, use the **remove** method. The following statement removes the button from the container:

```
container.remove(jbtOK);
```

When you run the program **MyFrameWithComponents**, the window will be displayed as in Figure 12.3(b). The button is always centered in the frame and occupies the entire frame no matter how you resize it. This is because components are put in the frame by the content pane's layout manager, and the default layout manager for the content pane places the button in the center. In the next section, you will use several different layout managers to place components in the desired locations.

12.5 Layout Managers

In many other window systems, the user-interface components are arranged by using hard-coded pixel measurements. For example, put a button at location (10, 10) in the window. Using hard-coded pixel measurements, the user interface might look fine on one system but be unusable on another. Java's layout managers provide a level of abstraction that automatically maps your user interface on all window systems.

The Java GUI components are placed in containers, where they are arranged by the container's layout manager. In the preceding program, you did not specify where to place the OK button in the frame, but Java knows where to place it, because the layout manager works behind the scenes to place components in the correct locations. A layout manager is created using a layout manager class.

Layout managers are set in containers using the `setLayout(aLayoutManager)` method. For example, you can use the following statements to create an instance of `XLayout` and set it in a container:

```
LayoutManager layoutManager = new XLayout();
container.setLayout(layoutManager);
```

This section introduces three basic layout managers: `FlowLayout`, `GridLayout`, and `BorderLayout`.

12.5.1 FlowLayout



Video Note
Use `FlowLayout`

`FlowLayout` is the simplest layout manager. The components are arranged in the container from left to right in the order in which they were added. When one row is filled, a new row is started. You can specify the way the components are aligned by using one of three constants: `FlowLayout.RIGHT`, `FlowLayout.CENTER`, or `FlowLayout.LEFT`. You can also specify the gap between components in pixels. The class diagram for `FlowLayout` is shown in Figure 12.4.

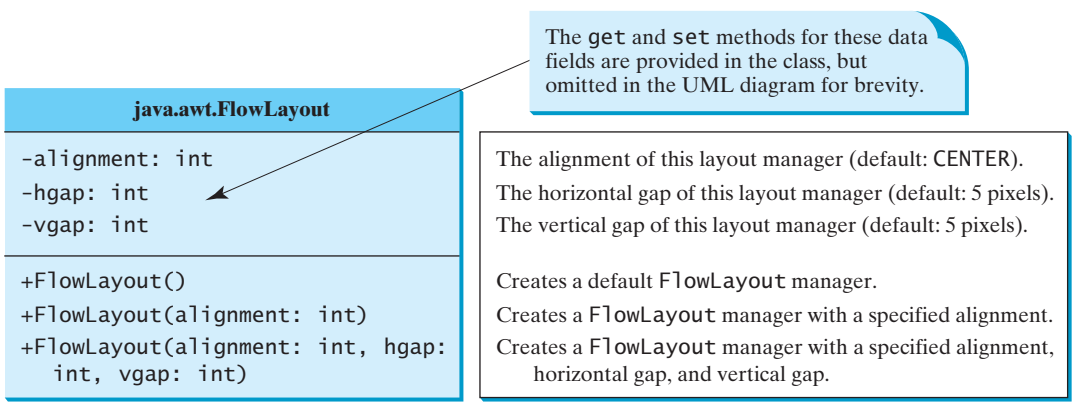

The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
java.awt.FlowLayout
-alignment: int
-hgap: int
-vgap: int
+FlowLayout()
+FlowLayout(alignment: int)
+FlowLayout(alignment: int, hgap: int, vgap: int)
The alignment of this layout manager (default: CENTER).
The horizontal gap of this layout manager (default: 5 pixels).
The vertical gap of this layout manager (default: 5 pixels).
Creates a default FlowLayout manager.
Creates a FlowLayout manager with a specified alignment.
Creates a FlowLayout manager with a specified alignment, horizontal gap, and vertical gap.

FIGURE 12.4 `FlowLayout` lays out components row by row.

Listing 12.3 gives a program that demonstrates flow layout. The program adds three labels and text fields into the frame with a `FlowLayout` manager, as shown in Figure 12.5.

LISTING 12.3 ShowFlowLayout.java

```
1 import javax.swing.JLabel;
2 import javax.swing.JTextField;
3 import javax.swing.JFrame;
```

extends **JFrame**

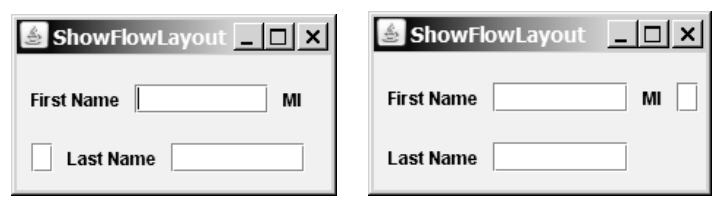
set layout

add label
add text field

create frame

set visible

```
 4 import java.awt.FlowLayout;
 5
 6 public class ShowFlowLayout extends JFrame {
 7   public ShowFlowLayout() {
 8     // Set FlowLayout, aligned left with horizontal gap 10
 9     // and vertical gap 20 between components
10     setLayout(new FlowLayout(FlowLayout.LEFT, 10, 20));
11
12     // Add labels and text fields to the frame
13     add(new JLabel("First Name"));
14     add(new JTextField(8));
15     add(new JLabel("MI"));
16     add(new JTextField(1));
17     add(new JLabel("Last Name"));
18     add(new JTextField(8));
19   }
20
21   /** Main method */
22   public static void main(String[] args) {
23     ShowFlowLayout frame = new ShowFlowLayout();
24     frame.setTitle("ShowFlowLayout");
25     frame.setSize(200, 200);
26     frame.setLocationRelativeTo(null); // Center the frame
27     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28     frame.setVisible(true);
29   }
30 }
```



(a) (b)

FIGURE 12.5 The components are added by the `FlowLayout` manager to fill in the rows in the container one after another.

This example creates a program using a style different from the programs in the preceding section, where frames were created using the `JFrame` class. This example creates a class named `ShowFlowLayout` that extends the `JFrame` class (line 6). The `main` method in this program creates an instance of `ShowFlowLayout` (line 23). The constructor of `ShowFlowLayout` constructs and places the components in the frame. This is the preferred style of creating GUI applications—for three reasons:

- Creating a GUI application means creating a frame, so it is natural to define a frame to extend `JFrame`.
- The frame may be further extended to add new components or functions.
- The class can be easily reused. For example, you can create multiple frames by creating multiple instances of the class.

Using one style consistently makes programs easy to read. From now on, most of the GUI main classes will extend the `JFrame` class. The constructor of the main class constructs the user interface. The `main` method creates an instance of the main class and then displays the frame.

In this example, the `FlowLayout` manager is used to place components in a frame. If you resize the frame, the components are automatically rearranged to fit. In Figure 12.5(a), the first row has three components, but in Figure 12.5(a), the first row has four components, because the width has been increased.

If you replace the `setLayout` statement (line 10) with `setLayout(new FlowLayout(FlowLayout.RIGHT, 0, 0))`, all the rows of buttons will be right aligned with no gaps.

An anonymous `FlowLayout` object was created in the statement (line 10):

```
setLayout(new FlowLayout(FlowLayout.LEFT, 10, 20));
```

which is equivalent to:

```
FlowLayout layout = new FlowLayout(FlowLayout.LEFT, 10, 20);
setLayout(layout);
```

This code creates an explicit reference to the object `layout` of the `FlowLayout` class. The explicit reference is not necessary, because the object is not directly referenced in the `ShowFlowLayout` class.

Suppose you add the same button into the frame ten times; will ten buttons appear in the frame? No, a GUI component such as a button can be added into only one container and only once in a container. Adding a button into a container multiple times is the same as adding it once.



Caution

Do not forget to put the `new` operator before a layout manager class when setting a layout style—for example, `setLayout(new FlowLayout())`.



Note

The constructor `ShowFlowLayout()` does not explicitly invoke the constructor `JFrame()`, but the constructor `JFrame()` is invoked implicitly. See §11.3.2, "Constructor Chaining."

12.5.2 GridLayout

The `GridLayout` manager arranges components in a grid (matrix) formation. The components are placed in the grid from left to right, starting with the first row, then the second, and so on, in the order in which they are added. The class diagram for `GridLayout` is shown in Figure 12.6.

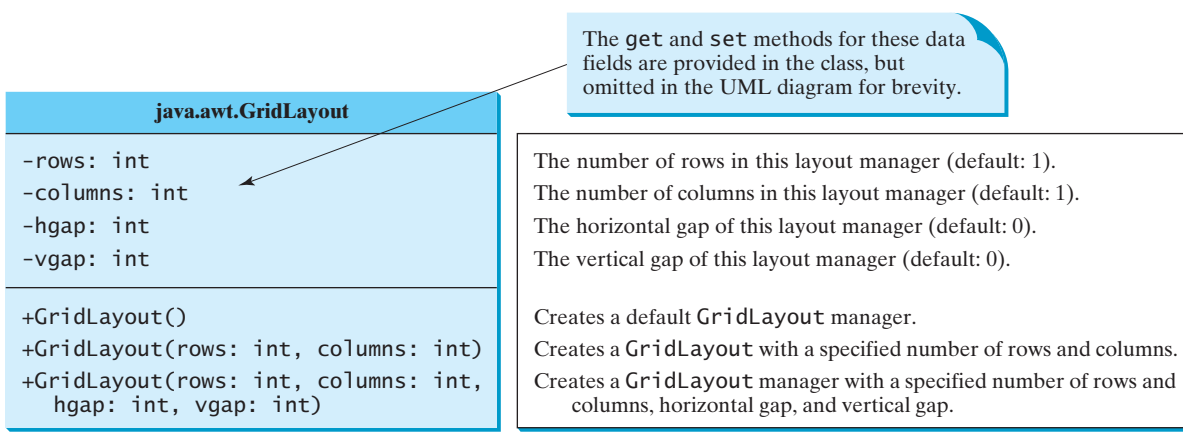

The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
java.awt.GridLayout
-rows: int
-columns: int
-hgap: int
-vgap: int
+GridLayout()
+GridLayout(rows: int, columns: int)
+GridLayout(rows: int, columns: int, hgap: int, vgap: int)
The number of rows in this layout manager (default: 1).
The number of columns in this layout manager (default: 1).
The horizontal gap of this layout manager (default: 0).
The vertical gap of this layout manager (default: 0).
Creates a default GridLayout manager.
Creates a GridLayout with a specified number of rows and columns.
Creates a GridLayout manager with a specified number of rows and columns, horizontal gap, and vertical gap.

FIGURE 12.6 `GridLayout` lays out components in equal-sized cells on a grid.

You can specify the number of rows and columns in the grid. The basic rule is as follows:

- The number of rows or the number of columns can be zero, but not both. If one is zero and the other is nonzero, the nonzero dimension is fixed, while the zero dimension is determined dynamically by the layout manager. For example, if you specify zero rows and three columns for a grid that has ten components, `GridLayout` creates three fixed columns of four rows, with the last row containing one component. If you specify three rows and zero columns for a grid that has ten components, `GridLayout` creates three fixed rows of four columns, with the last row containing two components.
- If both the number of rows and the number of columns are nonzero, the number of rows is the dominating parameter; that is, the number of rows is fixed, and the layout manager dynamically calculates the number of columns. For example, if you specify three rows and three columns for a grid that has ten components, `GridLayout` creates three fixed rows of four columns, with the last row containing two components.

Listing 12.4 gives a program that demonstrates grid layout. The program is similar to the one in Listing 12.3. It adds three labels and three text fields to the frame of `GridLayout` instead of `FlowLayout`, as shown in Figure 12.7.

LISTING 12.4 ShowGridLayout.java

```
1  import javax.swing.JLabel;
2  import javax.swing.JTextField;
3  import javax.swing.JFrame;
4  import java.awt.GridLayout;
5
6  public class ShowGridLayout extends JFrame {
7    public ShowGridLayout() {
8      // Set GridLayout, 3 rows, 2 columns, and gaps 5 between
9      // components horizontally and vertically
10     setLayout(new GridLayout(3, 2, 5, 5));
11
12     // Add labels and text fields to the frame
13     add(new JLabel("First Name"));
14     add(new JTextField(8));
15     add(new JLabel("MI"));
16     add(new JTextField(1));
17     add(new JLabel("Last Name"));
18     add(new JTextField(8));
19   }
20
21   /** Main method */
22   public static void main(String[] args) {
23     ShowGridLayout frame = new ShowGridLayout();
24     frame.setTitle("ShowGridLayout");
25     frame.setSize(200, 125);
26     frame.setLocationRelativeTo(null); // Center the frame
27     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28     frame.setVisible(true);
29   }
30 }
```

set layout
add label
add text field
create a frame
set visible

If you resize the frame, the layout of the buttons remains unchanged (i.e., the number of rows and columns does not change, and the gaps don't change either).

All components are given equal size in the container of `GridLayout`.

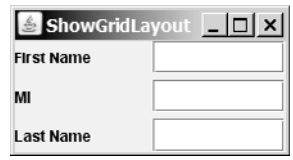


FIGURE 12.7 The `GridLayout` manager divides the container into grids; then the components are added to fill in the cells row by row.

Replacing the `setLayout` statement (line 10) with `setLayout(new GridLayout(3, 10))` would still yield three rows and *two* columns. The `columns` parameter is ignored because the `rows` parameter is nonzero. The actual number of columns is calculated by the layout manager.

What would happen if the `setLayout` statement (line 10) were replaced with `setLayout(new GridLayout(4, 2))` or with `setLayout(new GridLayout(2, 2))`? Please try it yourself.



Note

In `FlowLayout` and `GridLayout`, the order in which the components are added to the container is important. It determines the location of the components in the container.

12.5.3 BorderLayout

The `BorderLayout` manager divides a container into five areas: East, South, West, North, and Center. Components are added to a `BorderLayout` by using `add(Component, index)`, where `index` is a constant `BorderLayout.EAST`, `BorderLayout.SOUTH`, `BorderLayout.WEST`, `BorderLayout.NORTH`, or `BorderLayout.CENTER`. The class diagram for `BorderLayout` is shown in Figure 12.8.

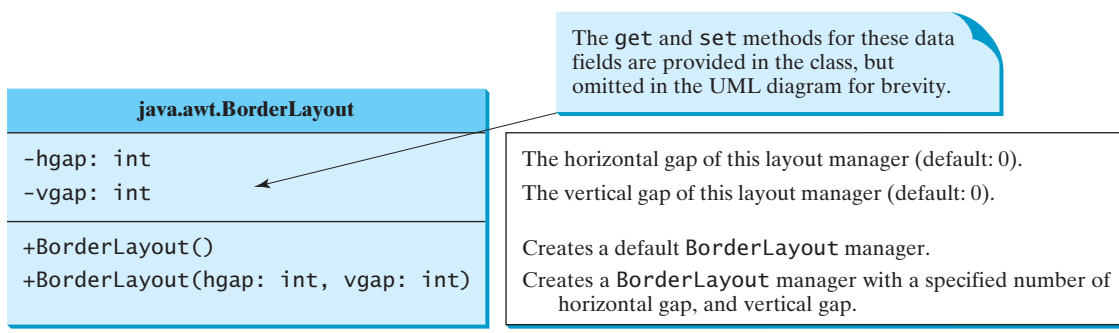

The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
java.awt.BorderLayout
-hgap: int
-vgap: int
+BorderLayout()
+BorderLayout(hgap: int, vgap: int)
The horizontal gap of this layout manager (default: 0).
The vertical gap of this layout manager (default: 0).
Creates a default BorderLayout manager.
Creates a BorderLayout manager with a specified number of horizontal gap, and vertical gap.

FIGURE 12.8 `BorderLayout` lays out components in five areas.

The components are laid out according to their preferred sizes and their placement in the container. The North and South components can stretch horizontally; the East and West components can stretch vertically; the Center component can stretch both horizontally and vertically to fill any empty space.

Listing 12.5 gives a program that demonstrates border layout. The program adds five buttons labeled `East`, `South`, `West`, `North`, and `Center` into the frame with a `BorderLayout` manager, as shown in Figure 12.9.

LISTING 12.5 ShowBorderLayout.java

```
1  import javax.swing.JButton;
2  import javax.swing.JFrame;
3  import java.awt.BorderLayout;
4
5  public class ShowBorderLayout extends JFrame {
6    public ShowBorderLayout() {
7      // Set BorderLayout with horizontal gap 5 and vertical gap 10
8      setLayout(new BorderLayout(5, 10));
9
10     // Add buttons to the frame
11     add(new JButton("East"), BorderLayout.EAST);
12     add(new JButton("South"), BorderLayout.SOUTH);
13     add(new JButton("West"), BorderLayout.WEST);
14     add(new JButton("North"), BorderLayout.NORTH);
15     add(new JButton("Center"), BorderLayout.CENTER);
16   }
17
18   /** Main method */
19   public static void main(String[] args) {
20     ShowBorderLayout frame = new ShowBorderLayout();
21     frame.setTitle("ShowBorderLayout");
22     frame.setSize(300, 200);
23     frame.setLocationRelativeTo(null); // Center the frame
24     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25     frame.setVisible(true);
26   }
27 }
```

set layout

add buttons

create a frame

set visible

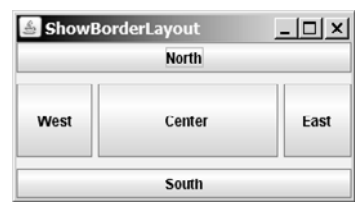


FIGURE 12.9 `BorderLayout` divides the container into five areas, each of which can hold a component.

The buttons are added to the frame (lines 11–15). Note that the `add` method for `BorderLayout` is different from the one for `FlowLayout` and `GridLayout`. With `BorderLayout` you specify where to put the components.

It is unnecessary to place components to occupy all the areas. If you remove the East button from the program and rerun it, you will see that the center stretches rightward to occupy the East area.



Note

`BorderLayout` interprets the absence of an index specification as `BorderLayout.CENTER`. For example, `add(component)` is the same as `add(Component, BorderLayout.CENTER)`. If you add two components into a container of `BorderLayout`, as follows,

```
container.add(component1);
container.add(component2);
```

only the last component is displayed.

12.5.4 Properties of Layout Managers

Layout managers have properties that can be changed dynamically. `FlowLayout` has `alignment`, `hgap`, and `vgap` properties. You can use the `setAlignment`, `setHgap`, and `setVgap` methods to specify the alignment and the horizontal and vertical gaps. `GridLayout` has the `rows`, `columns`, `hgap`, and `vgap` properties. You can use the `setRows`, `setColumns`, `setHgap`, and `setVgap` methods to specify the number of rows, the number of columns, and the horizontal and vertical gaps. `BorderLayout` has the `hgap` and `vgap` properties. You can use the `setHgap` and `setVgap` methods to specify the horizontal and vertical gaps.

In the preceding sections an anonymous layout manager is used because the properties of a layout manager do not change, once it is created. If you have to change the properties of a layout manager dynamically, the layout manager must be explicitly referenced by a variable. You can then change the properties of the layout manager through the variable. For example, the following code creates a layout manager and sets its properties:

```
// Create a layout manager
FlowLayout flowLayout = new FlowLayout();

// Set layout properties
flowLayout.setAlignment(FlowLayout.RIGHT);
flowLayout.setHgap(10);
flowLayout.setVgap(20);
```

12.6 Using Panels as Subcontainers



Video Note
Use panels as subcontainers

Suppose that you want to place ten buttons and a text field in a frame. The buttons are placed in grid formation, but the text field is placed on a separate row. It is difficult to achieve the desired look by placing all the components in a single container. With Java GUI programming, you can divide a window into panels. Panels act as subcontainers to group user-interface components. You add the buttons in one panel, then add the panel into the frame.

The Swing version of panel is `JPanel`. You can use `new JPanel()` to create a panel with a default `FlowLayout` manager or `new JPanel(LayoutManager)` to create a panel with the specified layout manager. Use the `add(Component)` method to add a component to the panel. For example, the following code creates a panel and adds a button to it:

```
JPanel p = new JPanel();
p.add(new JButton("OK"));
```

Panels can be placed inside a frame or inside another panel. The following statement places panel `p` into frame `f`:

```
f.add(p);
```

Listing 12.6 gives an example that demonstrates using panels as subcontainers. The program creates a user interface for a microwave oven, as shown in Figure 12.10.

LISTING 12.6 TestPanels.java

```
1 import java.awt.*;
2 import javax.swing.*;
3
4 public class TestPanels extends JFrame {
5   public TestPanels() {
6     // Create panel p1 for the buttons and set GridLayout
7     JPanel p1 = new JPanel();                                  panel p1
8     p1.setLayout(new GridLayout(4, 3));
9
```

```
10      // Add buttons to the panel
11      for (int i = 1; i <= 9; i++) {
12        p1.add(new JButton("" + i));
13      }
14
15      p1.add(new JButton("" + 0));
16      p1.add(new JButton("Start"));
17      p1.add(new JButton("Stop"));
18
19      // Create panel p2 to hold a text field and p1
20      JPanel p2 = new JPanel(new BorderLayout());
21      p2.add(new JTextField("Time to be displayed here"),
22        BorderLayout.NORTH);
23      p2.add(p1, BorderLayout.CENTER);
24
25      // add contents into the frame
26      add(p2, BorderLayout.EAST);
27      add(new JButton("Food to be placed here"),
28        BorderLayout.CENTER);
29    }
30
31    /** Main method */
32    public static void main(String[] args) {
33      TestPanels frame = new TestPanels();
34      frame.setTitle("The Front View of a Microwave Oven");
35      frame.setSize(400, 250);
36      frame.setLocationRelativeTo(null); // Center the frame
37      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
38      frame.setVisible(true);
39    }
40  }
```

panel p2

add p2 to frame

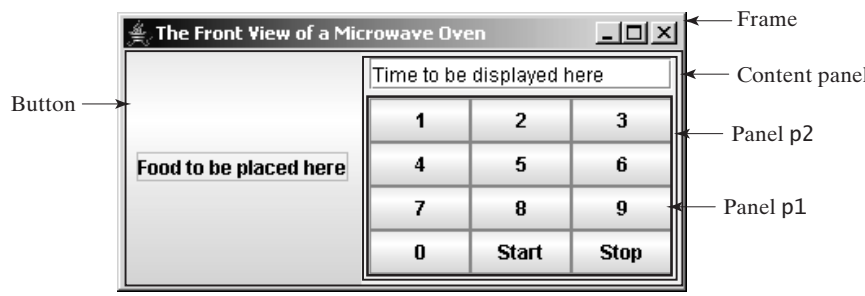


FIGURE 12.10 The program uses panels to organize components.

The `setLayout` method is defined in `java.awt.Container`. Since `JPanel` is a subclass of `Container`, you can use `setLayout` to set a new layout manager in the panel (line 8). Lines 7–8 can be replaced by `JPanel p1 = new JPanel(new GridLayout(4, 3))`.

To achieve the desired layout, the program uses panel `p1` of `GridLayout` to group the number buttons, the *Stop* button, and the *Start* button, and panel `p2` of `BorderLayout` to hold a text field in the north and `p1` in the center. The button representing the food is placed in the center of the frame, and `p2` is placed in the east of the frame.

The statement (lines 21–22)

```
p2.add(new JTextField("Time to be displayed here"),
  BorderLayout.NORTH);
```

creates an instance of `JTextField` and adds it to `p2`. `JTextField` is a GUI component that can be used for user input as well as to display values.



Note

It is worthwhile to note that the `Container` class is the superclass for GUI component classes, such as `JButton`. Every GUI component is a container. In theory, you could use the `setLayout` method to set the layout in a button and add components into a button, because all the public methods in the `Container` class are inherited into `JButton`, but for practical reasons you should not use buttons as containers.

superclass `Container`

12.7 The `Color` Class

You can set colors for GUI components by using the `java.awt.Color` class. Colors are made of red, green, and blue components, each represented by an `int` value that describes its intensity, ranging from `0` (darkest shade) to `255` (lightest shade). This is known as the *RGB model*.

You can create a color using the following constructor:

```
public Color(int r, int g, int b);
```

in which `r`, `g`, and `b` specify a color by its red, green, and blue components. For example,

```
Color color = new Color(128, 100, 100);
```



Note

The arguments `r`, `g`, `b` are between `0` and `255`. If a value beyond this range is passed to the argument, an `IllegalArgumentException` will occur.

`IllegalArgumentException`

You can use the `setBackground(Color c)` and `setForeground(Color c)` methods defined in the `java.awt.Component` class to set a component's background and foreground colors. Here is an example of setting the background and foreground of a button:

```
JButton jbtOK = new JButton("OK");
jbtOK.setBackground(color);
jbtOK.setForeground(new Color(100, 1, 1));
```

Alternatively, you can use one of the 13 standard colors (`BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE`, and `YELLOW`) defined as constants in `java.awt.Color`. The following code, for instance, sets the foreground color of a button to red:

```
jbtOK.setForeground(Color.RED);
```

12.8 The `Font` Class

You can create a font using the `java.awt.Font` class and set fonts for the components using the `setFont` method in the `Component` class.

The constructor for `Font` is:

```
public Font(String name, int style, int size);
```

You can choose a font name from `SansSerif`, `Serif`, `Monospaced`, `Dialog`, or `DialogInput`, choose a style from `Font.PLAIN` (`0`), `Font.BOLD` (`1`), `Font.ITALIC` (`2`), and `Font.BOLD + Font.ITALIC` (`3`), and specify a font size of any positive integer. For example, the following statements create two fonts and set one font to a button.

```
Font font1 = new Font("SansSerif", Font.BOLD, 16);
Font font2 = new Font("Serif", Font.BOLD + Font.ITALIC, 12);

JButton jbtOK = new JButton("OK");
jbtOK.setFont(font1);
```



Tip

find available fonts

If your system supports other fonts, such as "Times New Roman," you can use it to create a `Font` object. To find the fonts available on your system, you need to obtain an instance of `java.awt.GraphicsEnvironment` using its static method `getLocalGraphicsEnvironment()`. `GraphicsEnvironment` is an abstract class that describes the graphics environment on a particular system. You can use its `getAllFonts()` method to obtain all the available fonts on the system and its `getAvailableFontFamilyNames()` method to obtain the names of all the available fonts. For example, the following statements print all the available font names in the system:

```
GraphicsEnvironment e =
  GraphicsEnvironment.getLocalGraphicsEnvironment();
String[] fontnames = e.getAvailableFontFamilyNames();

for (int i = 0; i < fontnames.length; i++)
  System.out.println(fontnames[i]);
```

12.9 Common Features of Swing GUI Components

Component

In this chapter you have used several GUI components (e.g., `JFrame`, `Container`, `JPanel`, `JButton`, `JLabel`, `JTextField`). Many more GUI components will be introduced in this book. It is important to understand the common features of Swing GUI components. The



Video Note
Use Swing common properties

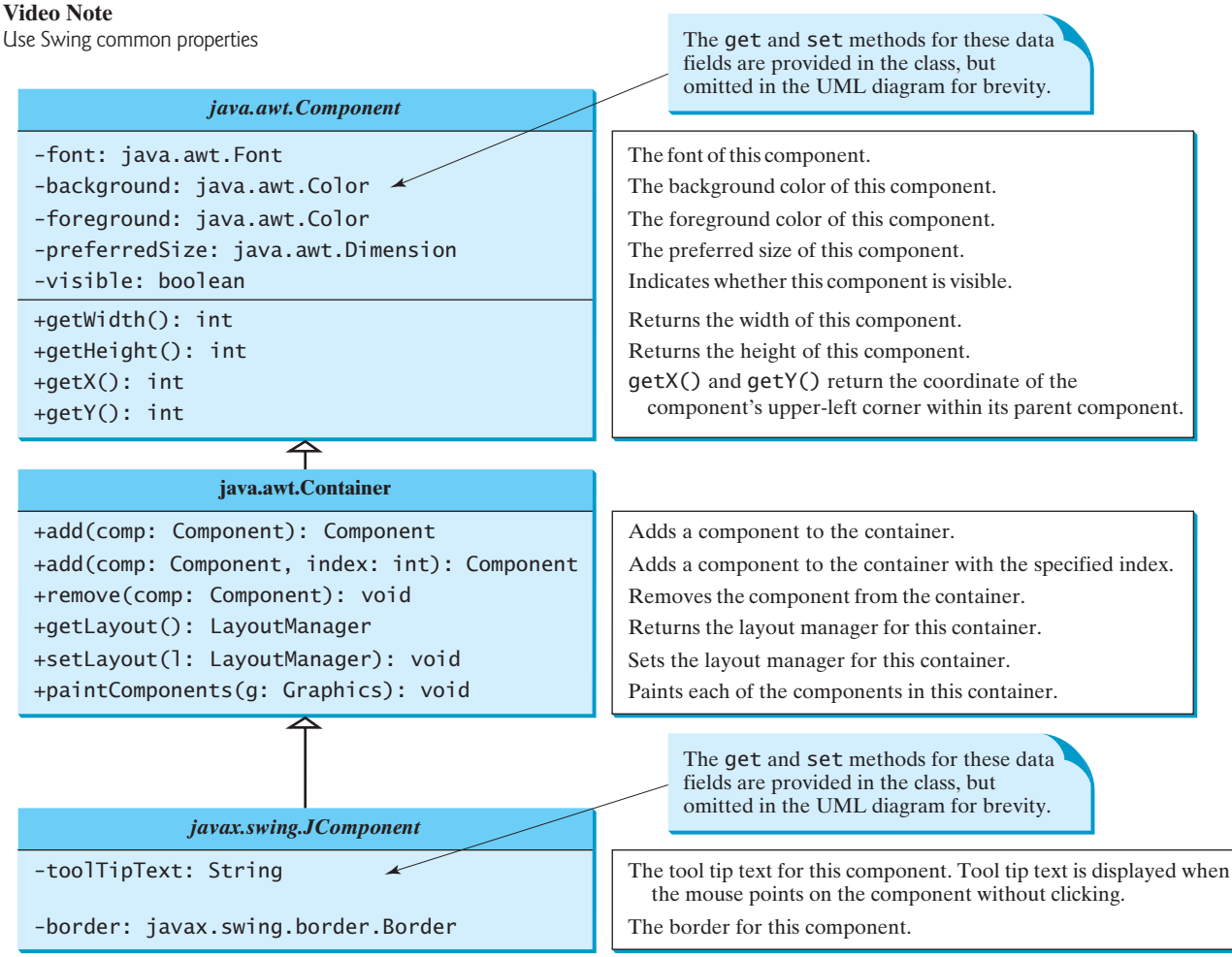

The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
java.awt.Component
-font: java.awt.Font
-background: java.awt.Color
-foreground: java.awt.Color
-preferredSize: java.awt.Dimension
-visible: boolean
+getWidth(): int
+getHeight(): int
+getX(): int
+getY(): int
The font of this component.
The background color of this component.
The foreground color of this component.
The preferred size of this component.
Indicates whether this component is visible.
Returns the width of this component.
Returns the height of this component.
getX() and getY() return the coordinate of the component's upper-left corner within its parent component.
java.awt.Container
+add(comp: Component): Component
+add(comp: Component, index: int): Component
+remove(comp: Component): void
+getLayout(): LayoutManager
+setLayout(l: LayoutManager): void
+paintComponents(g: Graphics): void
Adds a component to the container.
Adds a component to the container with the specified index.
Removes the component from the container.
Returns the layout manager for this container.
Sets the layout manager for this container.
Paints each of the components in this container.
The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
javax.swing.JComponent
-toolTipText: String
-border: javax.swing.border.Border
The tool tip text for this component. Tool tip text is displayed when the mouse points on the component without clicking.
The border for this component.

FIGURE 12.11 All the Swing GUI components inherit the public methods from `Component`, `Container`, and `JComponent`.

`Component` class is the root for all GUI components and containers. All Swing GUI components (except `JFrame`, `JApplet`, and `JDialog`) are subclasses of `JComponent`, as shown in Figure 12.1. Figure 12.11 lists some frequently used methods in `Component`, `Container`, and `JComponent` for manipulating properties such as font, color, size, tool tip text, and border.

`Container`

`JComponent`

A *tool tip* is text displayed on a component when you move the mouse on the component. It is often used to describe the function of a component.

You can set a border on any object of the `JComponent` class. Swing has several types of borders. To create a titled border, use `new TitledBorder(String title)`. To create a line border, use `new LineBorder(Color color, int width)`, where `width` specifies the thickness of the line.

Listing 12.7 is an example to demonstrate Swing common features. The example creates a panel `p1` to hold three buttons (line 8) and a panel `p2` to hold two labels (line 25), as shown in Figure 12.12. The background of the button `jbtLeft` is set to white (line 12) and the foreground of the button `jbtCenter` is set to green (line 13). The tool tip of the button `jbtRight` is set in line 14. Titled borders are set on panels `p1` and `p2` (lines 18, 36) and line borders are set on the labels (lines 32–33).

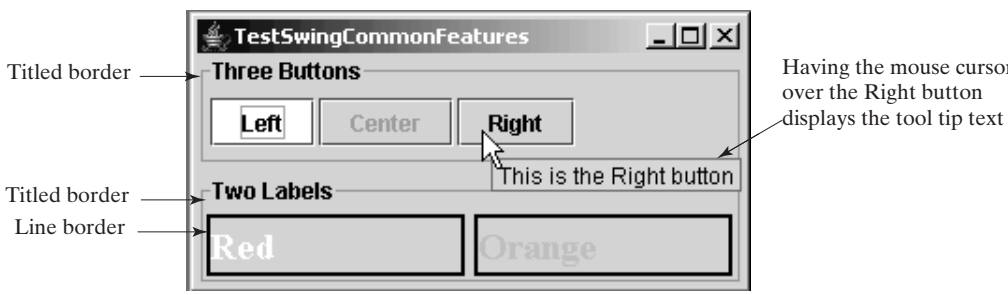


FIGURE 12.12 The font, color, border, and tool tip text are set in the message panel.

LISTING 12.7 TestSwingCommonFeatures.java

```
1  import java.awt.*;
2  import javax.swing.*;
3  import javax.swing.border.*;
4
5  public class TestSwingCommonFeatures extends JFrame {
6    public TestSwingCommonFeatures() {
7      // Create a panel to group three buttons
8      JPanel p1 = new JPanel(new FlowLayout(FlowLayout.LEFT, 2, 2));
9      JButton jbtLeft = new JButton("Left");
10     JButton jbtCenter = new JButton("Center");
11     JButton jbtRight = new JButton("Right");
12     jbtLeft.setBackground(Color.WHITE);                    set background
13     jbtCenter.setForeground(Color.GREEN);                  set foreground
14     jbtRight.setToolTipText("This is the Right button");   set tool tip text
15     p1.add(jbtLeft);
16     p1.add(jbtCenter);
17     p1.add(jbtRight);
18     p1.setBorder(new TitledBorder("Three Buttons"));       set titled border
19
20     // Create a font and a line border
21     Font largeFont = new Font("TimesRoman", Font.BOLD, 20);   create a font
22     Border lineBorder = new LineBorder(Color.BLACK, 2);       create a border
23
24     // Create a panel to group two labels
```

```
25      JPanel p2 = new JPanel(new GridLayout(1, 2, 5, 5));
26      JLabel jlblRed = new JLabel("Red");
27      JLabel jlblOrange = new JLabel("Orange");
28      jlblRed.setForeground(Color.RED);
29      jlblOrange.setForeground(Color.ORANGE);
30      jlblRed.setFont(largeFont);
31      jlblOrange.setFont(largeFont);
32      jlblRed.setBorder(lineBorder);
33      jlblOrange.setBorder(lineBorder);
34      p2.add(jlblRed);
35      p2.add(jlblOrange);
36      p2.setBorder(new TitledBorder("Two Labels"));
37
38      // Add two panels to the frame
39      setLayout(new GridLayout(2, 1, 5, 5));
40      add(p1);
41      add(p2);
42    }
43
44    public static void main(String[] args) {
45      // Create a frame and set its properties
46      JFrame frame = new TestSwingCommonFeatures();
47      frame.setTitle("TestSwingCommonFeatures");
48      frame.setSize(300, 150);
49      frame.setLocationRelativeTo(null); // Center the frame
50      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
51      frame.setVisible(true);
52    }
53  }
```

set foreground

set font

set line border

set titled border



Note

property default values

The same property may have different default values in different components. For example, the `visible` property in `JFrame` is `false` by default, but it is `true` in every instance of `JComponent` (e.g., `JButton` and `JLabel`) by default. To display a `JFrame`, you have to invoke `setVisible(true)` to set the `visible` property `true`, but you don't have to set this property for a `JButton` or a `JLabel`, because it is already `true`. To make a `JButton` or a `JLabel` invisible, you may invoke `setVisible(false)`. Please run the program and see the effect after inserting the following two statements in line 37:

```
jbtLeft.setVisible(false);
jlblRed.setVisible(false);
```

12.10 Image Icons

An icon is a fixed-size picture; typically it is small and used to decorate components. Images are normally stored in image files. Java currently supports three image formats: GIF (Graphics Interchange Format), JPEG (Joint Photographic Experts Group), and PNG (Portable Network Graphics). The image file names for these types end with .gif, .jpg, and .png, respectively. If you have a bitmap file or image files in other formats, you can use image-processing utilities to convert them into GIF, JPEG, or PNG format for use in Java.

image-file format

To display an image icon, first create an `ImageIcon` object using `new javax.swing.ImageIcon(filename)`. For example, the following statement creates an icon from an image file `us.gif` in the `image` directory under the current class path:

create **ImageIcon**

```
ImageIcon icon = new ImageIcon("image/us.gif");
```

"`image/us.gif`" is located in "`c:\book\image\us.gif`." The back slash (`\`) is the Windows file path notation. In Unix, the forward slash (`/`) should be used. In Java, the forward

file path character

slash (/) is used to denote a relative file path under the Java classpath (e.g., `image/us.gif`, as in this example).



Tip

File names are not case sensitive in Windows but are case sensitive in Unix. To enable your programs to run on all platforms, name all the image files consistently, using lowercase.

naming files consistently

An image icon can be displayed in a label or a button using `new JLabel(imageIcon)` or `new JButton(imageIcon)`. Listing 12.8 demonstrates displaying icons in labels and buttons. The example creates two labels and two buttons with icons, as shown in Figure 12.13.

LISTING 12.8 TestImageIcon.java

```
1 import javax.swing.*;
2 import java.awt.*;
3
4 public class TestImageIcon extends JFrame {
5   private ImageIcon usIcon = new ImageIcon("image/us.gif");
6   private ImageIcon myIcon = new ImageIcon("image/my.jpg");
7   private ImageIcon frIcon = new ImageIcon("image/fr.gif");
8   private ImageIcon ukIcon = new ImageIcon("image/uk.gif");
9
10  public TestImageIcon() {
11    setLayout(new GridLayout(1, 4, 5, 5));
12    add(new JLabel(usIcon));
13    add(new JLabel(myIcon));
14    add(new JButton(frIcon));
15    add(new JButton(ukIcon));
16  }
17
18  /** Main method */
19  public static void main(String[] args) {
20    TestImageIcon frame = new TestImageIcon();
21    frame.setTitle("TestImageIcon");
22    frame.setSize(200, 200);
23    frame.setLocationRelativeTo(null); // Center the frame
24    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25    frame.setVisible(true);
26  }
27 }
```

create image icons

a label with image

a button with image

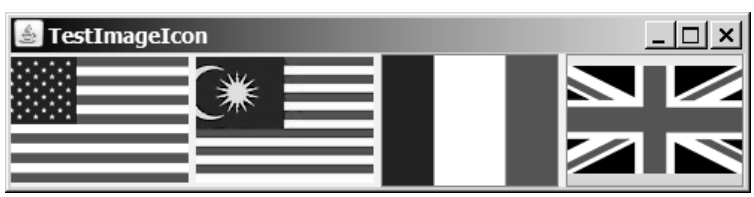


FIGURE 12.13 The image icons are displayed in labels and buttons.



Note

GUI components cannot be shared by containers, because one GUI component can appear in only one container at a time. Therefore, the relationship between a component and a container is the composition denoted by a solid diamond, as shown in Figure 12.1.

sharing borders and icons



Note

Borders and icons can be shared. Thus you can create a border or icon and use it to set the `border` or `icon` property for any GUI component. For example, the following statements set a border `b` for two panels `p1` and `p2`:

```
p1.setBorder(b);
p2.setBorder(b);
```

The following statements set an icon in two buttons `jbt1` and `jbt2`:

```
jbt1.setIcon(icon);
jbt2.setIcon(icon);
```

splash screen



Tip

A splash screen is an image that is displayed while the application is starting up. If your program takes a long time to load, you may display a splash screen to alert the user. For example, the following command:

```
java -splash:image/us.gif TestImageIcon
```

displays an image while the program `TestImageIcon` is being loaded.

Key Terms

AWT 406
heavyweight component 406
lightweight component 406
layout manager 411
Swing 406
splash screen 424

Chapter Summary

1. Every container has a layout manager that is used to position and place components in the container in the desired locations. Three simple and frequently used layout managers are `FlowLayout`, `GridLayout`, and `BorderLayout`.
2. You can use a `JPanel` as a subcontainer to group components to achieve a desired layout.
3. Use the `add` method to place components to a `JFrame` or a `JPanel`. By default, the frame's layout is `BorderLayout`, and the `JPanel`'s layout is `FlowLayout`.
4. You can set colors for GUI components by using the `java.awt.Color` class. Colors are made of red, green, and blue components, each represented by an unsigned byte value that describes its intensity, ranging from `0` (darkest shade) to `255` (lightest shade). This is known as the *RGB model*.
5. To create a `Color` object, use `new Color(r, g, b)`, in which `r`, `g`, and `b` specify a color by its red, green, and blue components. Alternatively, you can use one of the 13 standard colors (`BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE`, `YELLOW`) defined as constants in `java.awt.Color`.
6. Every Swing GUI component is a subclass of `javax.swing.JComponent`, and `JComponent` is a subclass of `java.awt.Component`. The properties `font`, `background`, `foreground`, `height`, `width`, and `preferredSize` in `Component` are inherited in these subclasses, as are `toolTipText` and `border` in `JComponent`.
7. You can use borders on any Swing components. You can create an image icon using the `ImageIcon` class and display it in a label and a button. Icons and borders can be shared.

REVIEW QUESTIONS

Sections 12.3–12.4

12.1 Which class is the root of the Java GUI component classes? Is a container class a subclass of `Component`? Which class is the root of the Swing GUI component classes?

12.2 Explain the difference between AWT GUI components, such as `java.awt.Button`, and Swing components, such as `javax.swing.JButton`.

12.3 How do you create a frame? How do you set the size for a frame? How do you get the size of a frame? How do you add components to a frame? What would happen if the statements `frame.setSize(400, 300)` and `frame.setVisible(true)` were swapped in Listing 12.2 `MyFrameWithComponents`?

12.4 Determine whether the following statements are true or false:

- You can add a button to a frame.
- You can add a frame to a panel.
- You can add a panel to a frame.
- You can add any number of components to a panel or a frame.
- You can derive a class from `JButton`, `JPanel`, or `JFrame`.

12.5 The following program is supposed to display a button in a frame, but nothing is displayed. What is the problem?

```
1 public class Test extends javax.swing.JFrame {
2   public Test() {
3     add(new javax.swing.JButton("OK"));
4   }
5
6   public static void main(String[] args) {
7     javax.swing.JFrame frame = new javax.swing.JFrame();
8     frame.setSize(100, 200);
9     frame.setVisible(true);
10  }
11 }
```

12.6 Which of the following statements have syntax errors?

```
Component c1 = new Component();
JComponent c2 = new JComponent();
Component c3 = new JButton();
JComponent c4 = new JButton();
Container c5 = new JButton();
c5.add(c4);
Object c6 = new JButton();
c5.add(c6);
```

Sections 12.5

12.7 Why do you need to use layout managers? What is the default layout manager for a frame? How do you add a component to a frame?

12.8 Describe `FlowLayout`. How do you create a `FlowLayout` manager? How do you add a component to a `FlowLayout` container? Is there a limit to the number of components that can be added to a `FlowLayout` container?

12.9 Describe `GridLayout`. How do you create a `GridLayout` manager? How do you add a component to a `GridLayout` container? Is there a limit to the number of components that can be added to a `GridLayout` container?

12.10 Describe `BorderLayout`. How do you create a `BorderLayout` manager? How do you add a component to a `BorderLayout` container?

Section 12.6

12.11 How do you create a panel with a specified layout manager?

12.12 What is the default layout manager for a `JPanel`? How do you add a component to a `JPanel`?

12.13 Can you use the `setTitle` method in a panel? What is the purpose of using a panel?

12.14 Since a GUI component class such as `JButton` is a subclass of `Container`, can you add components into a button?

Sections 12.7–12.8

12.15 How do you create a color? What is wrong about creating a `Color` using `new Color(400, 200, 300)`? Which of two colors is darker, `new Color(10, 0, 0)` or `new Color(200, 0, 0)`?

12.16 How do you create a font? How do you find all available fonts on your system?

Sections 12.9–12.10

12.17 How do you set background color, foreground color, font, and tool tip text on a Swing GUI component? Why is the tool tip text not displayed in the following code?

```
 1 import javax.swing.*;
 2
 3 public class Test extends JFrame {
 4   private JButton jbtOK = new JButton("OK");
 5
 6   public static void main(String[] args) {
 7     // Create a frame and set its properties
 8     JFrame frame = new Test();
 9     frame.setTitle("Logic Error");
10     frame.setSize(200, 100);
11     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12     frame.setVisible(true);
13   }
14
15   public Test() {
16     jbtOK.setToolTipText("This is a button");
17     add(new JButton("OK"));
18   }
19 }
```

12.18 Show the output of the following code:

```
import javax.swing.*;

public class Test {
  public static void main(String[] args) {
    JButton jbtOK = new JButton("OK");
    System.out.println(jbtOK.isVisible());

    JFrame frame = new JFrame();
    System.out.println(frame.isVisible());
  }
}
```

12.19 How do you create an `ImageIcon` from the file **image/us.gif** in the class directory?

12.20 What happens if you add a button to a container several times, as shown below? Does it cause syntax errors? Does it cause runtime errors?

```
JButton jbt = new JButton();
JPanel panel = new JPanel();
panel.add(jbt);
panel.add(jbt);
panel.add(jbt);
```

12.21 Will the following code display three buttons? Will the buttons display the same icon?

```
 1 import javax.swing.*;
 2 import java.awt.*;
 3
 4 public class Test extends JFrame  {
 5   public static void main(String[] args) {
 6     // Create a frame and set its properties
 7     JFrame frame = new Test();
 8     frame.setTitle("ButtonIcons");
 9     frame.setSize(200, 100);
10     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11     frame.setVisible(true);
12   }
13
14   public Test() {
15     ImageIcon usIcon = new ImageIcon("image/us.gif");
16     JButton jbt1 = new JButton(usIcon);
17     JButton jbt2 = new JButton(usIcon);
18
19     JPanel p1 = new JPanel();
20     p1.add(jbt1);
21
22     JPanel p2 = new JPanel();
23     p2.add(jbt2);
24
25     JPanel p3 = new JPanel();
26     p2.add(jbt1);
27
28     add(p1, BorderLayout.NORTH);
29     add(p2, BorderLayout.SOUTH);
30     add(p3, BorderLayout.CENTER);
31   }
32 }
```

12.22 Can a border or an icon be shared by GUI components?

Programming Exercises

Sections 12.5–12.6

12.1 (*Using the `FlowLayout` manager*) Write a program that meets the following requirements (see Figure 12.14):

- Create a frame and set its layout to `FlowLayout`.
- Create two panels and add them to the frame.
- Each panel contains three buttons. The panel uses `FlowLayout`.



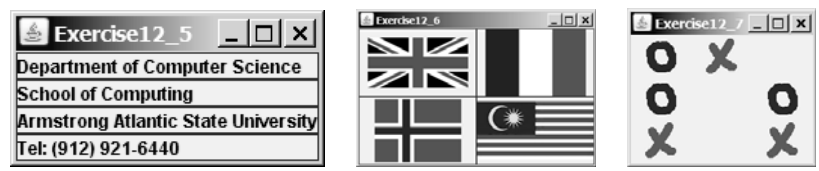
FIGURE 12.14 Exercise 12.1 places the first three buttons in one panel and the other three buttons in another panel.

12.2 (*Using the `BorderLayout` manager*) Rewrite the preceding program to create the same user interface, but instead of using `FlowLayout` for the frame, use `BorderLayout`. Place one panel in the south of the frame and the other in the center.

12.3 (*Using the `GridLayout` manager*) Rewrite the preceding program to create the same user interface. Instead of using `FlowLayout` for the panels, use a `GridLayout` of two rows and three columns.

12.4 (*Using `JPanel` to group buttons*) Rewrite the preceding program to create the same user interface. Instead of creating buttons and panels separately, define a class that extends the `JPanel` class. Place three buttons in your panel class, and create two panels from the user-defined panel class.

12.5 (*Displaying labels*) Write a program that displays four lines of text in four labels, as shown in Figure 12.15(a). Add a line border on each label.



(a) (b) (c)

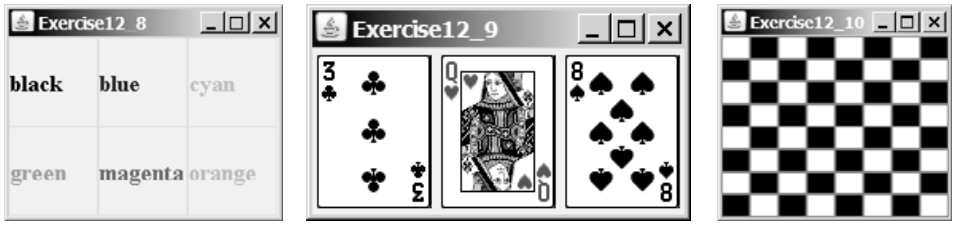
FIGURE 12.15 (a) Exercise 12.5 displays four labels. (b) Exercise 12.6 displays four icons. (c) Exercise 12.7 displays a TicTacToe board with image icons in labels.

Sections 12.7–12.10

12.6 (*Displaying icons*) Write a program that displays four icons in four labels, as shown in Figure 12.15(b). Add a line border on each label. (Use any images of your choice or those in the book, which can be obtained along with the book's source code.)

12.7** (*Game: displaying a TicTacToe board*) Display a frame that contains nine labels. A label may display an image icon for X, an image icon for O, or nothing, as shown in Figure 12.15(c). What to display is randomly decided. Use the `Math.random()` method to generate an integer `0`, `1`, or `2`, which corresponds to displaying a cross image icon, a not image icon, or nothing. The cross and not images are in the files `x.gif` and `o.gif`, which are under the image directory in www.cs.armstrong.edu/liang/intro8e/book.zip).

12.8* (*Swing common features*) Display a frame that contains six labels. Set the background of the labels to white. Set the foreground of the labels to black, blue, cyan, green, magenta, and orange, respectively, as shown in Figure 12.16(a). Set the border of each label to a line border with the yellow color. Set the font of each label to TimesRoman, bold, and 20 pixels. Set the text and tool tip text of each label to the name of its foreground color.



(a) (b) (c)

FIGURE 12.16 (a) Six labels are placed in the frame. (b) Three cards are randomly selected. (c) A checkerboard is displayed using buttons.

12.9* (*Game: displaying three cards*) Display a frame that contains three labels. Each label displays a card, as shown in Figure 12.16(b). The card image files are named 1.png, 2.png, . . . , 54.png and stored in the `image/card` directory. All three cards are distinct and selected randomly. The image files can be obtained from www.cs.armstrong.edu/liang/intro8e/book.zip.

12.10* (*Game: displaying a checkerboard*) Write a program that displays a checkerboard in which each white and black cell is a `JButton` with a background black or white, as shown in Figure 12.16(c).



Video Note
Display a checker board

This page intentionally left blank

CHAPTER 13

EXCEPTION HANDLING

Objectives

- To get an overview of exceptions and exception handling (§13.2).
- To explore the advantages of using exception handling (§13.3).
- To distinguish exception types: `Error` (fatal) vs. `Exception` (nonfatal) and checked vs. unchecked (§13.4).
- To declare exceptions in a method header (§13.5.1).
- To throw exceptions in a method (§13.5.2).
- To write a `try-catch` block to handle exceptions (§13.5.3).
- To explain how an exception is propagated (§13.5.3).
- To use the `finally` clause in a `try-catch` block (§13.6).
- To use exceptions only for unexpected errors (§13.7).
- To rethrow exceptions in a `catch` block (§13.8).
- To create chained exceptions (§13.9).
- To define custom exception classes (§13.10).



13.1 Introduction

Runtime errors occur while a program is running if the environment detects an operation that is impossible to carry out. For example, if you access an array using an index out of bounds, your program will get a runtime error with an `ArrayIndexOutOfBoundsException`. To read data from a file, you need to create a `Scanner` object using `new Scanner(new File(filename))` (see Listing 9.6). If the file does not exist, your program will get a runtime error with a `FileNotFoundException`.

In Java, runtime errors are caused by exceptions. An exception is an object that represents an error or a condition that prevents execution from proceeding normally. If the exception is not handled, the program will terminate abnormally. How can you handle the exception so that the program can continue to run or else terminate gracefully? This is the subject we introduce in this chapter.

13.2 Exception-Handling Overview



Video Note
Exception-handling advantages

To demonstrate exception handling, including how an exception object is created and thrown, we begin with an example (Listing 13.1) that reads in two integers and displays their quotient.

LISTING 13.1 Quotient.java

```
1 import java.util.Scanner;
2
3 public class Quotient {
4   public static void main(String[] args) {
5     Scanner input = new Scanner(System.in);
6
7     // Prompt the user to enter two integers
8     System.out.print("Enter two integers: ");
9     int number1 = input.nextInt();
10    int number2 = input.nextInt();
11
12    System.out.println(number1 + " / " + number2 + " is " +
13      (number1 / number2));
14  }
15 }
```

reads two integers

integer division



```
Enter two integers: 5 2 Enter
5 / 2 is 2
```



```
Enter two integers: 3 0 Enter
Exception in thread "main" java.lang.ArithmeticException: / by zero
                                at Quotient.main(Quotient.java:11)
```

If you entered `0` for the second number, a runtime error would occur, because you cannot divide an integer by `0`. (*Recall that a floating-point number divided by `0` does not raise an exception.*) A simple way to fix the error is to add an `if` statement to test the second number, as shown in Listing 13.2.

LISTING 13.2 QuotientWithIf.java

```
1 import java.util.Scanner;
2
3 public class QuotientWithIf {
```

```
 4   public static void main(String[] args) {
 5     Scanner input = new Scanner(System.in);
 6
 7     // Prompt the user to enter two integers
 8     System.out.print("Enter two integers: ");
 9     int number1 = input.nextInt();                          reads two integers
10     int number2 = input.nextInt();
11
12     if (number2 != 0)                                       test number2
13       System.out.println(number1 + " / " + number2
14         + " is " + (number1 / number2));
15     else
16       System.out.println("Divisor cannot be zero ");
17   }
18 }
```

```
Enter two integers: 5 0 Enter
Divisor cannot be zero
```



In order to demonstrate the concept of exception handling, including how to create, throw, catch, and handle an exception, we rewrite Listing 13.2 as shown in Listing 13.3.

LISTING 13.3 QuotientWithException.java

```
 1 import java.util.Scanner;
 2
 3 public class QuotientWithException {
 4   public static void main(String[] args) {
 5     Scanner input = new Scanner(System.in);
 6
 7     // Prompt the user to enter two integers
 8     System.out.print("Enter two integers: ");
 9     int number1 = input.nextInt();                          reads two integers
10     int number2 = input.nextInt();
11
12     try {                                                   try block
13       if (number2 == 0)
14         throw new ArithmeticException("Divisor cannot be zero");
15
16       System.out.println(number1 + " / " + number2 + " is " +
17         (number1 / number2));
18     }
19     catch (ArithmeticException ex) {                        catch block
20       System.out.println("Exception: an integer " +
21         "cannot be divided by zero ");
22     }
23
24     System.out.println("Execution continues ...");
25   }
26 }
```

```
Enter two integers: 5 3 Enter
5 / 3 is 1
Execution continues ...
```





```
Enter two integers: 5 0 ↵Enter
Exception: an integer cannot be divided by zero
Execution continues ...
```

The program contains a `try` block and a `catch` block. The `try` block (lines 12–18) contains the code that is executed in normal circumstances. The `catch` block (lines 19–22) contains the code that is executed when `number2` is `0`. In this event the program throws an exception by executing

`throw` statement

```
throw new ArithmeticException("Divisor cannot be zero");
```

The value thrown, in this case `new ArithmeticException("Divisor cannot be zero")`, is called an *exception*. The execution of a `throw` statement is called *throwing an exception*. The exception is an object created from an exception class. In this case, the exception class is `java.lang.ArithmeticException`.

exception

throwing exception

When an exception is thrown, the normal execution flow is interrupted. As the name suggests, to "throw an exception" is to pass the exception from one place to another. The exception is caught by the `catch` block. The code in the `catch` block is executed to *handle the exception*. Afterward, the statement (line 24) after the `catch` block is executed.

handle exception

The `throw` statement is analogous to a method call, but instead of calling a method, it calls a `catch` block. In this sense, a `catch` block is like a method definition with a parameter that matches the type of the value being thrown. Unlike a method, after the `catch` block is executed, however, the program control does not return to the `throw` statement; instead, it executes the next statement after the `catch` block.

The identifier `ex` in the `catch`–block header

```
catch (ArithmeticException ex)
```

acts very much like a parameter in a method. So this parameter is referred to as a `catch`–block parameter. The type (e.g., `ArithmeticException`) preceding `ex` specifies what kind of exception the `catch` block can catch. Once the exception is caught, you can access the thrown value from this parameter in the body of a catch block.

`catch`–block parameter

In summary, a template for a `try`-`throw`-`catch` block may look like this:

```
try {
  Code to try;
  Throw an exception with a throw statement or
    from method if necessary;
  More code to try;
}
catch (type ex) {
  Code to process the exception;
}
```

An exception may be thrown directly by using a `throw` statement in a `try` block, or by invoking a method that may throw an exception.

13.3 Exception-Handling Advantages

You have seen from Listing 13.3 how an exception is created, thrown, caught, and handled. You may wonder what the benefits are. To see these benefits, we rewrite Listing 13.3 to compute a quotient using a method, as shown in Listing 13.4.

LISTING 13.4 QuotientWithMethod.java

```
1 import java.util.Scanner;
2
3 public class QuotientWithMethod {
4   public static int quotient(int number1, int number2) {
5     if (number2 == 0)
6       throw new ArithmeticException("Divisor cannot be zero");
7
8     return number1 / number2;
9   }
10
11  public static void main(String[] args) {
12    Scanner input = new Scanner(System.in);
13
14    // Prompt the user to enter two integers
15    System.out.print("Enter two integers: ");
16    int number1 = input.nextInt();
17    int number2 = input.nextInt();
18
19    try {
20      int result = quotient(number1, number2);
21      System.out.println(number1 + " / " + number2 + " is "
22        + result);
23    }
24    catch (ArithmeticException ex) {
25      System.out.println("Exception: an integer " +
26        "cannot be divided by zero ");
27    }
28
29    System.out.println("Execution continues ...");
30  }
31 }
```

If an ArithmeticException occurs → line 25

quotient method

throw exception

reads two integers

try block

invoke method

catch block

```
Enter two integers: 5 3 Enter
5 / 3 is 1
Execution continues ...
```



```
Enter two integers: 5 0 Enter
Exception: an integer cannot be divided by zero
Execution continues ...
```



Method **quotient** (lines 4–9) returns the quotient of two integers. If **number2** is **0**, it cannot return a value. So, an exception is thrown in line 6.

The main method invokes **quotient** (line 20). If the quotient method executes normally, it returns a value to the caller. If the **quotient** method encounters an exception, it throws the exception back to its caller. The caller's **catch** block handles the exception.

Now you see the *advantages* of using exception handling. It enables a method to throw an exception to its caller. The caller can handle this exception. Without this capability, the called method itself must handle the exception or terminate the program. Often the called method does not know what to do in case of error. This is typically the case for the library methods. The library method can detect the error, but only the caller knows what needs to be done when

advantage

an error occurs. The essential benefit of exception handling is to separate the detection of an error (done in a called method) from the handling of an error (done in the calling method).

Many library methods *throw exceptions*. Listing 13.5 gives an example that handles `FileNotFoundException` for invoking the `Scanner(File file)` constructor.

LISTING 13.5 FileNotFoundExceptionDemo.java

```
1  import java.util.Scanner;
2  import java.io.*;
3
4  public class FileNotFoundExceptionDemo {
5    public static void main(String[] args) {
6      Scanner inputFromConsole = new Scanner(System.in);
7      // Prompt the user to enter a file name
8      System.out.print("Enter a file name: ");
9      String filename = inputFromConsole.nextLine();
10
11     try {
12       Scanner inputFromFile = new Scanner(new File(filename));
13       System.out.println("File " + filename + " exists ");
14       // Processing file ...
15     }
16     catch (FileNotFoundException ex) {
17       System.out.println("Exception: " +  filename + " not found");
18     }
19   }
20 }
```

try block create a Scanner

If a FileNotFoundException occurs

catch block



```
Enter a file name: c:\book\Welcome.java Enter
File c:\book\Welcome.java exists
```



```
Enter a file name: c:\book\Test10.java Enter
Exception: c:\book\Test10.java not found
```

The program creates a `Scanner` for a file (line 12). If the file does not exist, the constructor throws a `FileNotFoundException`, which is caught in the `catch` block.

Listing 13.6 gives an example that handles an `InputMismatchException` exception.

LISTING 13.6 InputMismatchExceptionDemo.java

```
1  import java.util.*;
2
3  public class InputMismatchExceptionDemo {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6      boolean continueInput = true;
7
8      do {
9        try {
10         System.out.print("Enter an integer: ");
11         int number = input.nextInt();
12
13         // Display the result
14         System.out.println(
```

try block create a Scanner

If an InputMismatchException occurs

```
15            "The number entered is " + number);
16
17          continueInput = false;
18        }
19        catch (InputMismatchException ex) {
20          System.out.println("Try again. (" +
21            "Incorrect input: an integer is required)");
22          input.nextLine(); // Discard input
23        }
24      } while (continueInput);
25    }
26  }
```

catch block

```
Enter an integer: 3.5 Enter
Try again. (Incorrect input: an integer is required)
Enter an integer: 4 Enter
The number entered is 4
```



When executing `input.nextInt()` (line 11), an `InputMismatchException` occurs if the input entered is not an integer. Suppose `3.5` is entered. An `InputMismatchException` occurs and the control is transferred to the `catch` block. The statements in the `catch` block are now executed. The statement `input.nextLine()` in line 22 discards the current input line so that the user can enter a new line of input. The variable `continueInput` controls the loop. Its initial value is `true` (line 6), and it is changed to `false` (line 17) when a valid input is received.

13.4 Exception Types

The preceding sections used `ArithmeticException`, `FileNotFoundException`, and `InputMismatchException`. Are there any other types of exceptions you can use? Yes. There are many predefined exception classes in the Java API. Figure 13.1 shows some of them.

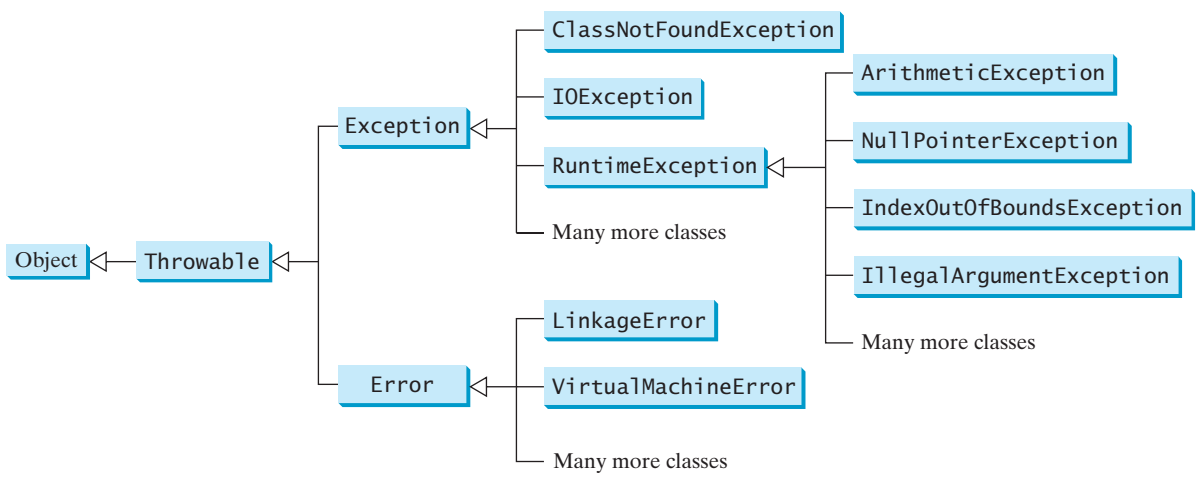


FIGURE 13.1 Exceptions thrown are instances of the classes shown in this diagram, or of subclasses of one of these classes.



Note

The class names `Error`, `Exception`, and `RuntimeException` are somewhat confusing. All three of these classes are exceptions, and all of the errors discussed here occur at runtime.

The `Throwable` class is the root of exception classes. All Java exception classes inherit directly or indirectly from `Throwable`. You can create your own exception classes by extending `Exception` or a subclass of `Exception`.

The exception classes can be classified into three major types: system errors, exceptions, and runtime exceptions.

system error

- *System errors* are thrown by the JVM and represented in the `Error` class. The `Error` class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully. Examples of subclasses of `Error` are listed in Table 13.1.

TABLE 13.1 Examples of Subclasses of `Error`

Class	*Possible Reason for Exception*
`LinkageError`	A class has some dependency on another class, but the latter class has changed incompatibly after the compilation of the former class.
`VirtualMachineError`	The JVM is broken or has run out of the resources it needs in order to continue operating.

exception

- *Exceptions* are represented in the `Exception` class, which describes errors caused by your program and by external circumstances. These errors can be caught and handled by your program. Examples of subclasses of `Exception` are listed in Table 13.2.

TABLE 13.2 Examples of Subclasses of `Exception`

Class	*Possible Reason for Exception*
`ClassNotFoundException`	Attempt to use a class that does not exist. This exception would occur, for example, if you tried to run a nonexistent class using the **java** command, or if your program were composed of, say, three class files, only two of which could be found.
`IOException`	Related to input/output operations, such as invalid input, reading past the end of a file, and opening a nonexistent file. Examples of subclasses of `IOException` are `InterruptedIOException`, `EOFException` (EOF is short for End Of File), and `FileNotFoundException`.

runtime exception

- *Runtime exceptions* are represented in the `RuntimeException` class, which describes programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors. Runtime exceptions are generally thrown by the JVM. Examples of subclasses are listed in Table 13.3.

TABLE 13.3 Examples of Subclasses of `RuntimeException`

Class	*Possible Reason for Exception*
`ArithmeticException`	Dividing an integer by zero. Note that floating-point arithmetic does not throw exceptions. See Appendix E, "Special Floating-Point Values."
`NullPointerException`	Attempt to access an object through a `null` reference variable.
`IndexOutOfBoundsException`	Index to an array is out of range.
`IllegalArgumentException`	A method is passed an argument that is illegal or inappropriate.

`RuntimeException`, `Error`, and their subclasses are known as *unchecked exceptions*. All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with them.

unchecked exception
checked exception

In most cases, unchecked exceptions reflect programming logic errors that are unrecoverable. For example, a `NullPointerException` is thrown if you access an object through a reference variable before an object is assigned to it; an `IndexOutOfBoundsException` is thrown if you access an element in an array outside the bounds of the array. These are logic errors that should be corrected in the program. Unchecked exceptions can occur anywhere in a program. To avoid cumbersome overuse of `try-catch` blocks, Java does not mandate that you write code to catch or declare unchecked exceptions.



Caution

At present, Java does not throw integer overflow or underflow exceptions. The following statement adds `1` to the maximum integer.

integer overflow/underflow

```
int number = Integer.MAX_VALUE + 1;
System.out.println(number);
```

It displays `-2147483648`, which is logically incorrect. The cause of this problem is overflow; that is, the result exceeds the maximum for an `int` value.

A future version of Java may fix this problem by throwing an overflow exception.

13.5 More on Exception Handling

The preceding sections gave you an overview of exception handling and introduced several predefined exception types. This section provides an in-depth discussion of exception handling.

Java's exception-handling model is based on three operations: *declaring an exception*, *throwing an exception*, and *catching an exception*, as shown in Figure 13.2.

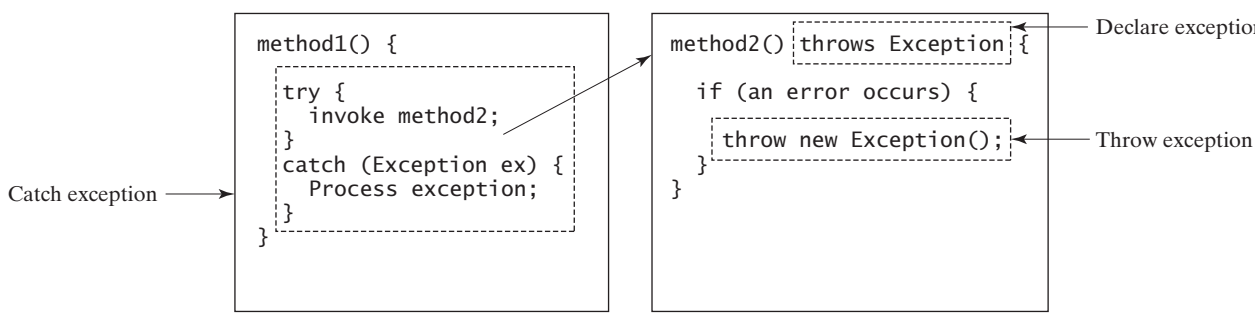


FIGURE 13.2 Exception handling in Java consists of declaring exceptions, throwing exceptions, and catching and processing exceptions.

13.5.1 Declaring Exceptions

In Java, the statement currently being executed belongs to a method. The Java interpreter invokes the `main` method to start executing a program. Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*. Because system errors and runtime errors can happen to any code, Java does not require that you declare `Error` and `RuntimeException` (unchecked exceptions) explicitly in the method. However, all other exceptions thrown by the method must be explicitly declared in the method header so that the caller of the method is informed of the exception.

declare exception

To declare an exception in a method, use the `throws` keyword in the method header, as in this example:

```
public void myMethod() throws IOException
```

The `throws` keyword indicates that `myMethod` might throw an `IOException`. If the method might throw multiple exceptions, add a list of the exceptions, separated by commas, after `throws`:

```
public void myMethod()
  throws Exception1, Exception2, ..., ExceptionN
```



Note

If a method does not declare exceptions in the superclass, you cannot override it to declare exceptions in the subclass.

13.5.2 Throwing Exceptions

throw exception

A program that detects an error can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example: Suppose the program detects that an argument passed to the method violates the method contract (e.g., the argument must be nonnegative, but a negative argument is passed); the program can create an instance of `IllegalArgumentException` and throw it, as follows:

```
IllegalArgumentException ex =
  new IllegalArgumentException("Wrong Argument");
throw ex;
```

Or, if you prefer, you can use the following:

```
throw new IllegalArgumentException("Wrong Argument");
```



Note

`IllegalArgumentException` is an exception class in the Java API. In general, each exception class in the Java API has at least two constructors: a no-arg constructor, and a constructor with a `String` argument that describes the exception. This argument is called the *exception message*, which can be obtained using `getMessage()`.

exception message



Tip

The keyword to declare an exception is `throws`, and the keyword to throw an exception is `throw`.

throws and **throw**

13.5.3 Catching Exceptions

catch exception

You now know how to declare an exception and how to throw an exception. When an exception is thrown, it can be caught and handled in a try-catch block, as follows:

```
try {
  statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
```

```
catch (ExceptionN exVar3) {
  handler for exceptionN;
}
```

If no exceptions arise during the execution of the `try` block, the `catch` blocks are skipped.

If one of the statements inside the `try` block throws an exception, Java skips the remaining statements in the try block and starts the process of finding the code to handle the exception. The code that handles the exception is called the *exception handler*; it is found by propagating the exception backward through a chain of method calls, starting from the current method. Each `catch` block is examined in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the `catch` block. If so, the exception object is assigned to the variable declared, and the code in the `catch` block is executed. If no handler is found, Java exits this method, passes the exception to the method that invoked the method, and continues the same process to find a handler. If no handler is found in the chain of methods being invoked, the program terminates and prints an error message on the console. The process of finding a handler is called *catching an exception*.

exception handler

Suppose the `main` method invokes `method1`, `method1` invokes `method2`, `method2` invokes `method3`, and `method3` throws an exception, as shown in Figure 13.3. Consider the following scenario:

- If the exception type is `Exception3`, it is caught by the `catch` block for handling exception `ex3` in `method2`. `statement5` is skipped, and `statement6` is executed.
- If the exception type is `Exception2`, `method2` is aborted, the control is returned to `method1`, and the exception is caught by the `catch` block for handling exception `ex2` in `method1`. `statement3` is skipped, and `statement4` is executed.
- If the exception type is `Exception1`, `method1` is aborted, the control is returned to the `main` method, and the exception is caught by the `catch` block for handling exception `ex1` in the `main` method. `statement1` is skipped, and `statement2` is executed.
- If the exception type is not caught in `method2`, `method1`, and `main`, the program terminates. `statement1` and `statement2` are not executed.

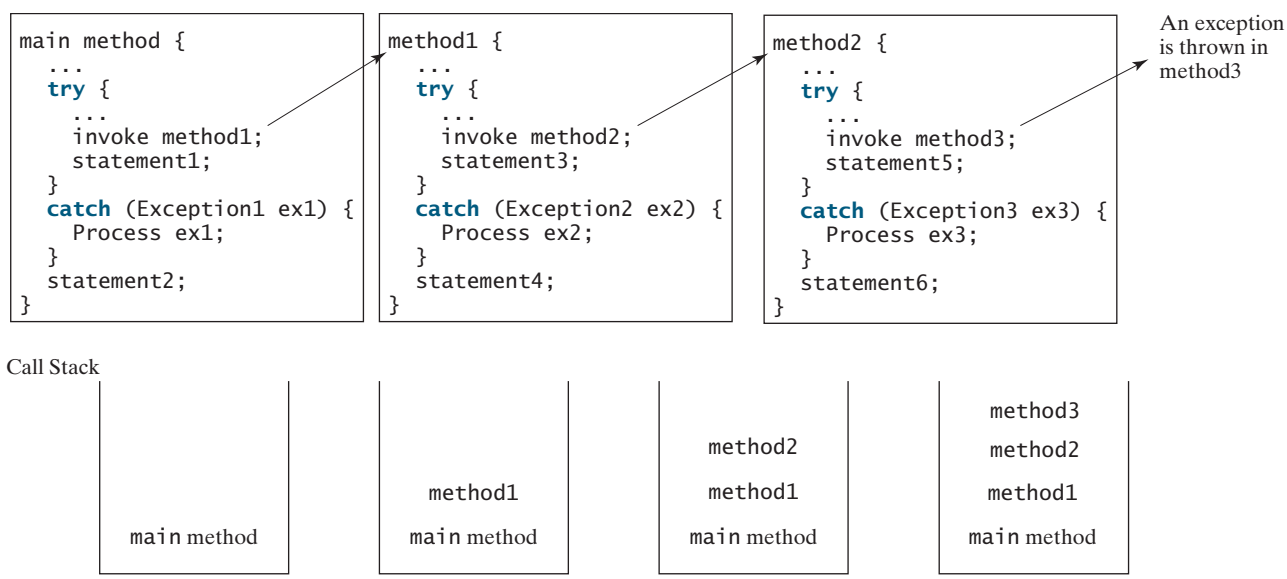


FIGURE 13.3 If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the `main` method.

catch block



Note

Various exception classes can be derived from a common superclass. If a `catch` block catches exception objects of a superclass, it can catch all the exception objects of the subclasses of that superclass.

order of exception handlers



Note

The order in which exceptions are specified in `catch` blocks is important. A compile error will result if a catch block for a superclass type appears before a catch block for a subclass type. For example, the ordering in (a) below is erroneous, because `RuntimeException` is a subclass of `Exception`. The correct ordering should be as shown in (b).

```
try {
  ...
}
catch (Exception ex) {
  ...
}
catch (RuntimeException ex) {
  ...
}
```

(a) Wrong order

```
try {
  ...
}
catch (RuntimeException ex) {
  ...
}
catch (Exception ex) {
  ...
}
```

(b) Correct order

catch or declare checked exceptions



Note

Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than `Error` or `RuntimeException`), you must invoke it in a `try-catch` block or declare to throw the exception in the calling method. For example, suppose that method `p1` invokes method `p2`, and `p2` may throw a checked exception (e.g., `IOException`); you have to write the code as shown in (a) or (b) below.

```
void p1() {
  try {
    p2();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a) Catch exception

```
void p1() throws IOException {

  p2();

}
```

(b) Throw exception

13.5.4 Getting Information from Exceptions

methods in `Throwable`

An exception object contains valuable information about the exception. You may use the following instance methods in the `java.lang.Throwable` class to get information regarding the exception, as shown in Figure 13.4. The `printStackTrace()` method prints stack trace

java.lang.Throwable	
+getMessage(): String	Returns the message of this object.
+toString(): String	Returns the concatenation of three strings: (1) the full name of the exception class; (2) ":" (a colon and a space); (3) the `getMessage()` method.
+printStackTrace(): void	Prints the `Throwable` object and its call stack trace information on the console.
+getStackTrace(): StackTraceElement[]	Returns an array of stack trace elements representing the stack trace pertaining to this throwable.

FIGURE 13.4 `Throwable` is the root class for all exception objects.

information on the console. The `getStackTrace()` method provides programmatic access to the stack trace information printed by `printStackTrace()`.

Listing 13.7 gives an example that uses the methods in `Throwable` to display exception information. Line 4 invokes the `sum` method to return the sum of all the elements in the array. There is an error in line 23 that causes the `ArrayIndexOutOfBoundsException`, a subclass of `IndexOutOfBoundsException`. This exception is caught in the try-catch block. Lines 7, 8, 9 display the stack trace, exception message, and exception object and message using the `printStackTrace()`, `getMessage()`, and `toString()` methods, as shown in Figure 13.5. Line 10 brings stack trace elements into an array. Each element represents a method call. You can obtain the method (line 12), class name (line 13), and exception line number (line 14) for each element.

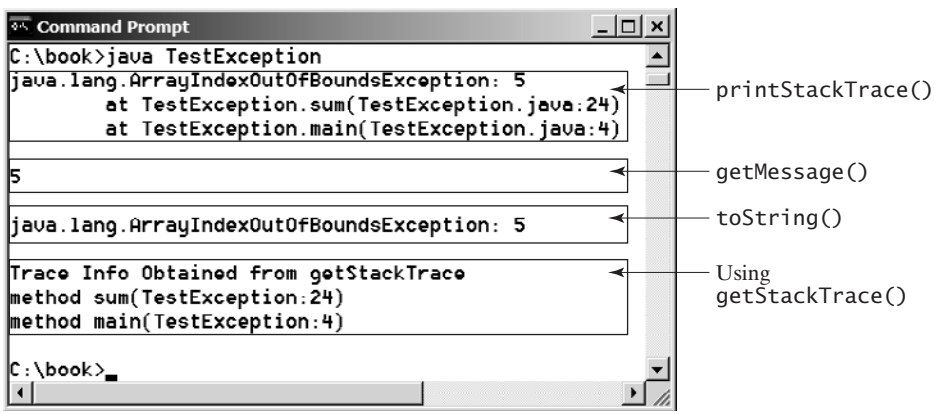


FIGURE 13.5 You can use the `printStackTrace()`, `getMessage()`, `toString()`, and `getStackTrace()` methods to obtain information from exception objects.

LISTING 13.7 TestException.java

```
1  public class TestException  {
2    public static void main(String[] args) {
3      try {
4        System.out.println(sum(new int[] {1, 2, 3, 4, 5}));          invoke sum
5      }
6      catch (Exception ex) {
7        ex.printStackTrace();                                        printStackTrace()
8        System.out.println("\n" + ex.getMessage());                  getMessage()
9        System.out.println("\n" + ex.toString());                    toString()
10
11       System.out.println("\nTrace Info Obtained from getStackTrace");
12       StackTraceElement[] traceElements = ex.getStackTrace();
13       for (int i = 0; i < traceElements.length; i++) {
14         System.out.print("method " + traceElements[i].getMethodName());
15         System.out.print("(" + traceElements[i].getClassName() + ":");
16         System.out.println(traceElements[i].getLineNumber() + ")");
17       }
18     }
19   }
20
21   private static int sum(int[] list) {
22     int result = 0;
23     for (int i = 0; i <= list.length; i++)
24       result += list[i];
25     return result;
26   }
27 }
```

13.5.5 Example: Declaring, Throwing, and Catching Exceptions

This example demonstrates declaring, throwing, and catching exceptions by modifying the `setRadius` method in the `Circle` class in Listing 8.9, Circle3.java. The new `setRadius` method throws an exception if the radius is negative.

Rename the circle class given in Listing 13.8 as `CircleWithException`, which is the same as `Circle3` except that the `setRadius(double newRadius)` method throws an `IllegalArgumentException` if the argument `newRadius` is negative.

LISTING 13.8 CircleWithException.java

```
1 public class CircleWithException {
2   /** The radius of the circle */
3   private double radius;
4
5   /** The number of the objects created */
6   private static int numberOfObjects = 0;
7
8   /** Construct a circle with radius 1 */
9   public CircleWithException() {
10    this(1.0);
11  }
12
13  /** Construct a circle with a specified radius */
14  public CircleWithException(double newRadius) {
15    setRadius(newRadius);
16    numberOfObjects++;
17  }
18
19  /** Return radius */
20  public double getRadius() {
21    return radius;
22  }
23
24  /** Set a new radius */
25  public void setRadius(double newRadius)
26      throws IllegalArgumentException {
27    if (newRadius >= 0)
28      radius =  newRadius;
29    else
30      throw new IllegalArgumentException(
31        "Radius cannot be negative");
32  }
33
34  /** Return numberOfObjects */
35  public static int getNumberOfObjects() {
36    return numberOfObjects;
37  }
38
39  /** Return the area of this circle */
40  public double findArea() {
41    return radius * radius * 3.14159;
42  }
43 }
```

declare exception

throw exception

A test program that uses the new `Circle` class is given in Listing 13.9.

LISTING 13.9 TestCircleWithException.java

```
1 public class TestCircleWithException {
2   public static void main(String[] args) {
```

```
 3     try {
 4       CircleWithException c1 = new CircleWithException(5);
 5       CircleWithException c2 = new CircleWithException(-5);
 6       CircleWithException c3 = new CircleWithException(0);
 7     }
 8     catch (IllegalArgumentException ex) {
 9       System.out.println(ex);
10     }
11
12     System.out.println("Number of objects created: " +
13       CircleWithException.getNumberOfObjects());
14   }
15 }
```

try

catch

```
java.lang.IllegalArgumentException: Radius cannot be negative
Number of objects created: 1
```



The original `Circle` class remains intact except that the class name is changed to `CircleWithException`, a new constructor `CircleWithException(newRadius)` is added, and the `setRadius` method now declares an exception and throws it if the radius is negative.

The `setRadius` method declares to throw `IllegalArgumentException` in the method header (lines 25–32 in `CircleWithException.java`). The `CircleWithException` class would still compile if the `throws IllegalArgumentException` clause were removed from the method declaration, since it is a subclass of `RuntimeException` and every method can throw `RuntimeException` (unchecked exception) regardless of whether it is declared in the method header.

The test program creates three `CircleWithException` objects, `c1`, `c2`, and `c3`, to test how to handle exceptions. Invoking `new CircleWithException(-5)` (line 5 in Listing 13.9) causes the `setRadius` method to be invoked, which throws an `IllegalArgumentException`, because the radius is negative. In the `catch` block, the type of the object `ex` is `IllegalArgumentException`, which matches the exception object thrown by the `setRadius` method. So, this exception is caught by the `catch` block.

The exception handler prints a short message, `ex.toString()` (line 9), about the exception, using `System.out.println(ex)`.

Note that the execution continues in the event of the exception. If the handlers had not caught the exception, the program would have abruptly terminated.

The test program would still compile if the `try` statement were not used, because the method throws an instance of `IllegalArgumentException`, a subclass of `RuntimeException` (unchecked exception). If a method throws an exception other than `RuntimeException` and `Error`, the method must be invoked within a `try-catch` block.

13.6 The `finally` Clause

Occasionally, you may want some code to be executed regardless of whether an exception occurs or is caught. Java has a `finally` clause that can be used to accomplish this objective. The syntax for the `finally` clause might look like this:

```
try {
  statements;
}
catch (TheException ex) {
  handling ex;
}
```

```
finally {
  finalStatements;
}
```

The code in the `finally` block is executed under all circumstances, regardless of whether an exception occurs in the `try` block or is caught. Consider three possible cases:

- If no exception arises in the `try` block, `finalStatements` is executed, and the next statement after the `try` statement is executed.
- If a statement causes an exception in the `try` block that is caught in a `catch` block, the rest of statements in the `try` block are skipped, the `catch` block is executed, and the `finally` clause is executed. The next statement after the `try` statement is executed.
- If one of the statements causes an exception that is not caught in any `catch` block, the other statements in the `try` block are skipped, the `finally` clause is executed, and the exception is passed to the caller of this method.

The `finally` block executes even if there is a `return` statement prior to reaching the `finally` block.



Note

omitting catch block

The `catch` block may be omitted when the `finally` clause is used.

A common use of the `finally` clause is in I/O programming. To ensure that a file is closed under all circumstances, you may place a file closing statement in the `finally` block, as shown in Listing 13.10.

LISTING 13.10 FinallyDemo.java

```
1  public class FinallyDemo {
2    public static void main(String[] args) {
3      java.io.PrintWriter output = null;
4
5      try {
6        // Create a file
7        output = new java.io.PrintWriter("text.txt");
8
9        // Write formatted output to the file
10       output.println("Welcome to Java");
11     }
12     catch (java.io.IOException ex) {
13       ex.printStackTrace();
14     }
15     finally {
16       // Close the file
17       if (output != null) output.close();
18     }
19
20     System.out.println("End of program");
21   }
22 }
```

try

catch

finally

The statements in lines 7 and 10 may throw an `IOException`, so they are placed inside a `try` block. The statement `output.close()` closes the `PrintWriter` object output in the `finally` block. This statement is executed regardless of whether an exception occurs in the `try` block or is caught.

13.7 When to Use Exceptions

The `try` block contains the code that is executed in normal circumstances. The `catch` block contains the code that is executed in exceptional circumstances. Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify. Be aware, however, that exception handling usually requires more time and resources, because it requires instantiating a new exception object, rolling back the call stack, and propagating the exception through the chain of methods invoked to search for the handler.

An exception occurs in a method. If you want the exception to be processed by its caller, you should create an exception object and throw it. If you can handle the exception in the method where it occurs, there is no need to throw or use exceptions.

In general, common exceptions that may occur in multiple classes in a project are candidates for exception classes. Simple errors that may occur in individual methods are best handled locally without throwing exceptions.

When should you use a try-catch block in the code? Use it when you have to deal with unexpected error conditions. Do not use a try-catch block to deal with simple, expected situations. For example, the following code

```
try {
  System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
  System.out.println("refVar is null");
}
```

is better replaced by

```
if (refVar != null)
  System.out.println(refVar.toString());
else
  System.out.println("refVar is null");
```

Which situations are exceptional and which are expected is sometimes difficult to decide. The point is not to abuse exception handling as a way to deal with a simple logic test.

13.8 Rethrowing Exceptions

Java allows an exception handler to rethrow the exception if the handler cannot process the exception or simply wants to let its caller be notified of the exception. The syntax may look like this:

```
try {
  statements;
}
catch (TheException ex) {
  perform operations before exits;
  throw ex;
}
```

The statement `throw ex` rethrows the exception to the caller so that other handlers in the caller get a chance to process the exception `ex`.

13.9 Chained Exceptions

In the preceding section, the catch block rethrows the original exception. Sometimes, you may need to throw a new exception (with additional information) along with the original exception.

This is called *chained exceptions*. Listing 13.11 illustrates how to create and throw chained exceptions.

LISTING 13.11 ChainedExceptionDemo.java

stack trace

chained exception

throw exception

```
1  public class ChainedExceptionDemo {
2    public static void main(String[] args) {
3      try {
4        method1();
5      }
6      catch (Exception ex) {
7        ex.printStackTrace();
8      }
9    }
10
11   public static void method1() throws Exception {
12     try {
13       method2();
14     }
15     catch (Exception ex) {
16       throw new Exception("New info from method1", ex);
17     }
18   }
19
20   public static void method2() throws Exception {
21     throw new Exception("New info from method2");
22   }
23 }
```



```
java.lang.Exception: New info from method1
       at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:16)
       at ChainedExceptionDemo.main(ChainedExceptionDemo.java:4)
Caused by: java.lang.Exception: New info from method2
       at ChainedExceptionDemo.method2(ChainedExceptionDemo.java:21)
       at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:13)
       ... 1 more
```

The `main` method invokes `method1` (line 4), `method1` invokes `method2` (line 13), and `method2` throws an exception (line 21). This exception is caught in the `catch` block in `method1` and is wrapped in a new exception in line 16. The new exception is thrown and caught in the `catch` block in the `main` method in line 4. The sample output shows the output from the `printStackTrace()` method in line 7. The new exception thrown from `method1` is displayed first, followed by the original exception thrown from `method2`.

13.10 Creating Custom Exception Classes



Video Note
Create custom exception classes

Java provides quite a few exception classes. Use them whenever possible instead of creating your own exception classes. However, if you run into a problem that cannot be adequately described by the predefined exception classes, you can create your own exception class, derived from `Exception` or from a subclass of `Exception`, such as `IOException`.

In Listing 13.8, CircleWithException.java, the `setRadius` method throws an exception if the radius is negative. Suppose you wish to pass the radius to the handler. In that case, you may create a custom exception class, as shown in Listing 13.12.

LISTING 13.12 InvalidRadiusException.java

```
1 public class InvalidRadiusException extends Exception {
2   private double radius;
3
4   /** Construct an exception */
5   public InvalidRadiusException(double radius) {
6     super("Invalid radius " + radius);
7     this.radius = radius;
8   }
9
10  /** Return the radius */
11  public double getRadius() {
12    return radius;
13  }
14 }
```

extends **Exception**

This custom exception class extends **java.lang.Exception** (line 1). The **Exception** class extends **java.lang.Throwable**. All the methods (e.g., **getMessage()**, **toString()**, and **printStackTrace()**) in **Exception** are inherited from **Throwable**. The **Exception** class contains four constructors. Among them, the following two constructors are often used:

java.lang.Exception	
+Exception()	Constructs an exception with no message.
+Exception(message: String)	Constructs an exception with the specified message.

Line 6 invokes the superclass's constructor with a message. This message will be set in the exception object and can be obtained by invoking **getMessage()** on the object.



Tip

Most exception classes in the Java API contain two constructors: a no-arg constructor and a constructor with a message parameter.

To create an **InvalidRadiusException**, you have to pass a radius. So, the **setRadius** method in Listing 13.8 can be modified as follows:

```
/** Set a new radius */
public void setRadius(double newRadius)
    throws InvalidRadiusException {
  if (newRadius >= 0)
    radius =  newRadius;
  else
    throw new InvalidRadiusException(newRadius);
}
```

The following code creates a circle object and sets its radius to **-5**.

```
try {
  CircleWithException1 c = new CircleWithException1(4);
  c.setRadius(-5);
}
catch (InvalidRadiusException ex) {
  System.out.println("The invalid radius is " + ex.getRadius());
}
```

Invoking `setRadius(-5)` throws an `InvalidRadiusException`, which is caught by the handler. The handler displays the radius in the exception object `ex`.

checked custom exception



Tip

Can you define a custom exception class by extending `RuntimeException`? Yes, but it is not a good way to go, because it makes your custom exception unchecked. It is better to make a custom exception checked, so that the complier can force these exceptions to be caught in your program.

Key Terms

chained exception 448
checked exception 438
declare exception 439
exception 438
exception propagation 441
throw exception 436
unchecked exception 438

Chapter Summary

1. Exception handling enables a method to throw an exception to its caller.
2. A Java exception is an instance of a class derived from `java.lang.Throwable`. Java provides a number of predefined exception classes, such as `Error`, `Exception`, `RuntimeException`, `ClassNotFoundException`, `NullPointerException`, and `ArithmeticException`. You can also define your own exception class by extending `Exception`.
3. Exceptions occur during the execution of a method. `RuntimeException` and `Error` are unchecked exceptions; all other exceptions are checked.
4. When declaring a method, you have to declare a checked exception if the method might throw it, thus telling the compiler what can go wrong.
5. The keyword for declaring an exception is `throws`, and the keyword for throwing an exception is `throw`.
6. To invoke the method that declares checked exceptions, you must enclose the method call in a `try` statement. When an exception occurs during the execution of the method, the `catch` block catches and handles the exception.
7. If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the `main` method.
8. Various exception classes can be derived from a common superclass. If a `catch` block catches the exception objects of a superclass, it can also catch all the exception objects of the subclasses of that superclass.
9. The order in which exceptions are specified in a `catch` block is important. A compile error will result if you do not specify an exception object of a class before an exception object of the superclass of that class.
10. When an exception occurs in a method, the method exits immediately if it does not catch the exception. If the method is required to perform some task before exiting, you can catch the exception in the method and then rethrow it to the real handler.
11. The code in the `finally` block is executed under all circumstances, regardless of whether an exception occurs in the `try` block or is caught.

12. Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.

13. Exception handling should not be used to replace simple tests. You should test simple exceptions whenever possible, and reserve exception handling for dealing with situations that cannot be handled with `if` statements.

Review Questions

Sections 13.1–13.3

13.1 Describe the Java `Throwable` class, its subclasses, and the types of exceptions. What `RuntimeException` will the following programs throw, if any?

```
public class Test {
  public static void main(String[] args) {
    System.out.println(1 / 0);
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    int[] list = new int[5];
    System.out.println(list[5]);
  }
}
```

(b)

```
public class Test {
  public static void main(String[] args) {
    String s = "abc";
    System.out.println(s.charAt(3));
  }
}
```

(c)

```
public class Test {
  public static void main(String[] args) {
    Object o = new Object();
    String d = (String)o;
  }
}
```

(d)

```
public class Test {
  public static void main(String[] args) {
    Object o = null;
    System.out.println(o.toString());
  }
}
```

(e)

```
public class Test {
  public static void main(String[] args) {
    System.out.println(1.0 / 0);
  }
}
```

(f)

13.2 Show the output of the following code.

```
public class Test {
  public static void main(String[] args) {
    for (int i = 0; i < 2; i++) {
      System.out.print(i + " ");
      try {
        System.out.println(1 / 0);
      }
      catch (Exception ex) {
      }
    }
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    try {
      for (int i = 0; i < 2; i++) {
        System.out.print(i + " ");
        System.out.println(1 / 0);
      }
    }
    catch (Exception ex) {
    }
  }
}
```

(b)

13.3 Point out the problem in the following code. Does the code throw any exceptions?

```
long value = Long.MAX_VALUE + 1;
System.out.println(value);
```

13.4 What is the purpose of declaring exceptions? How do you declare an exception, and where? Can you declare multiple exceptions in a method header?

13.5 What is a checked exception, and what is an unchecked exception?

13.6 How do you throw an exception? Can you throw multiple exceptions in one `throw` statement?

13.7 What is the keyword `throw` used for? What is the keyword `throws` used for?

13.8 What does the JVM do when an exception occurs? How do you catch an exception?

13.9 What is the printout of the following code?

```
public class Test {
  public static void main(String[] args) {
    try {
      int value = 30;
      if (value < 40)
        throw new Exception("value is too small");
    }
    catch (Exception ex) {
      System.out.println(ex.getMessage());
    }
    System.out.println("Continue after the catch block");
  }
}
```

What would be the printout if the line

```
int value = 30;
```

were changed to

```
int value = 50;
```

13.10 Suppose that `statement2` causes an exception in the following `try-catch` block:

```
try {
  statement1;
  statement2;
  statement3;
}
catch (Exception1 ex1) {
}
catch (Exception2 ex2) {
}

statement4;
```

Answer the following questions:

- Will `statement3` be executed?
- If the exception is not caught, will `statement4` be executed?
- If the exception is caught in the `catch` block, will `statement4` be executed?
- If the exception is passed to the caller, will `statement4` be executed?

13.11 What is displayed when the following program is run?

```
public class Test {
  public static void main(String[] args) {
```

```
      try {
        int[] list = new int[10];
        System.out.println("list[10] is " + list[10]);
      }
      catch (ArithmeticException ex) {
        System.out.println("ArithmeticException");
      }
      catch (RuntimeException ex) {
        System.out.println("RuntimeException");
      }
      catch (Exception ex) {
        System.out.println("Exception");
      }
    }
  }
```

13.12 What is displayed when the following program is run?

```
public class Test {
  public static void main(String[] args) {
    try {
      method();
      System.out.println("After the method call");
    }
    catch (ArithmeticException ex) {
      System.out.println("ArithmeticException");
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException");
    }
    catch (Exception e) {
      System.out.println("Exception");
    }
  }

  static void method() throws Exception {
    System.out.println(1 / 0);
  }
}
```

13.13 What is displayed when the following program is run?

```
public class Test {
  public static void main(String[] args) {
    try {
      method();
      System.out.println("After the method call");
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException in main");
    }
    catch (Exception ex) {
      System.out.println("Exception in main");
    }
  }

  static void method() throws Exception {
    try {
      String s ="abc";
      System.out.println(s.charAt(3));
    }
```

```
    catch (RuntimeException ex) {
      System.out.println("RuntimeException in method()");
    }
    catch (Exception ex) {
      System.out.println("Exception in method()");
    }
  }
}
```

13.14 What does the method `getMessage()` do?

13.15 What does the method `printStackTrace` do?

13.16 Does the presence of a `try-catch` block impose overhead when no exception occurs?

13.17 Correct a compile error in the following code:

```
public void m(int value) {
  if (value < 40)
    throw new Exception("value is too small");
}
```

Sections 13.4–13.10

13.18 Suppose that `statement2` causes an exception in the following statement:

```
try {
  statement1;
  statement2;
  statement3;
}
catch (Exception1 ex1) {
}
catch (Exception2 ex2) {
}
catch (Exception3 ex3) {
  throw ex3;
}
finally {
  statement4;
};
statement5;
```

Answer the following questions:

- Will `statement5` be executed if the exception is not caught?
- If the exception is of type `Exception3`, will `statement4` be executed, and will `statement5` be executed?

13.19 Suppose the `setRadius` method throws the `RadiusException` defined in §13.7. What is displayed when the following program is run?

```
public class Test {
  public static void main(String[] args) {
    try {
      method();
      System.out.println("After the method call");
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException in main");
    }
```

```
    catch (Exception ex) {
      System.out.println("Exception in main");
    }
  }

  static void method() throws Exception {
    try {
      Circle c1 = new Circle(1);
      c1.setRadius(-1);
      System.out.println(c1.getRadius());
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException in method()");
    }
    catch (Exception ex) {
      System.out.println("Exception in method()");
      throw ex;
    }
  }
}
```

13.20 The following method checks whether a string is a numeric string:

```
public static boolean isNumeric(String token) {
  try {
    Double.parseDouble(token);
    return true;
  }
  catch (java.lang.NumberFormatException ex) {
    return false;
  }
}
```

Is it correct? Rewrite it without using exceptions.

Programming Exercises

Sections 13.2–13.10

13.1* (*NumberFormatException*) Listing 9.5, Calculator.java, is a simple command-line calculator. Note that the program terminates if any operand is nonnumeric. Write a program with an exception handler that deals with nonnumeric operands; then write another program without using an exception handler to achieve the same objective. Your program should display a message that informs the user of the wrong operand type before exiting (see Figure 13.6).

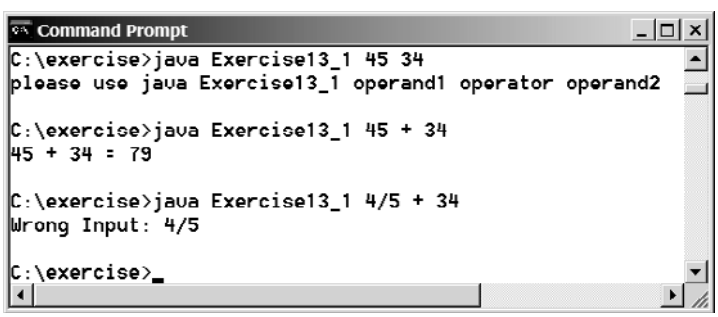

Command Prompt
C:\exercise>java Exercise13_1 45 34
please use java Exercise13_1 operand1 operator operand2
C:\exercise>java Exercise13_1 45 + 34
45 + 34 = 79
C:\exercise>java Exercise13_1 4/5 + 34
Wrong Input: 4/5
C:\exercise>

Figure 13.6 The program performs arithmetic operations and detects input errors.

13.2* (*NumberFormatException*) Write a program that prompts the user to read two integers and displays their sum. Your program should prompt the user to read the number again if the input is incorrect.

13.3* (*ArrayIndexOutBoundsException*) Write a program that meets the following requirements:

- Create an array with 100 randomly chosen integers.
- Prompt the user to enter the index of the array, then display the corresponding element value. If the specified index is out of bounds, display the message **Out of Bounds**.

13.4* (*IllegalArgumentException*) Modify the `Loan` class in Listing 10.2 to throw `IllegalArgumentException` if the loan amount, interest rate, or number of years is less than or equal to zero.

13.5* (*IllegalTriangleException*) Exercise 11.1 defined the `Triangle` class with three sides. In a triangle, the sum of any two sides is greater than the other side. The `Triangle` class must adhere to this rule. Create the `IllegalTriangleException` class, and modify the constructor of the `Triangle` class to throw an `IllegalTriangleException` object if a triangle is created with sides that violate the rule, as follows:

```
/** Construct a triangle with the specified sides */
public Triangle(double side1, double side2, double side3)
  throws IllegalTriangleException {
  // Implement it
}
```

13.6* (*NumberFormatException*) Listing 9.2 implements the `hexToDecimal(String hexString)` method, which converts a hex string into a decimal number. Implement the `hexToDecimal` method to throw a `NumberFormatException` if the string is not a hex string.

13.7* (*NumberFormatException*) Exercise 9.8 specifies the `binaryToDecimal(String binaryString)` method, which converts a binary string into a decimal number. Implement the `binaryToDecimal` method to throw a `NumberFormatException` if the string is not a binary string.



Video Note
`HexFormatException`

13.8* (*HexFormatException*) Exercise 13.6 implements the `hexToDecimal` method to throw a `NumberFormatException` if the string is not a hex string. Define a custom exception called `HexFormatException`. Implement the `hexToDecimal` method to throw a `HexFormatException` if the string is not a hex string.

13.9* (*BinaryFormatException*) Exercise 13.7 implements the `binaryToDecimal` method to throw a `BinaryFormatException` if the string is not a binary string. Define a custom exception called `BinaryFormatException`. Implement the `binaryToDecimal` method to throw a `BinaryFormatException` if the string is not a binary string.

13.10* (*OutOfMemoryError*) Write a program that causes the JVM to throw an `OutOfMemoryError` and catches and handles this error.

CHAPTER 14

ABSTRACT CLASSES AND INTERFACES

Objectives

- To design and use abstract classes (§14.2).
- To process a calendar using the `Calendar` and `GregorianCalendar` classes (§14.3).
- To specify common behavior for objects using interfaces (§14.4).
- To define interfaces and define classes that implement interfaces (§14.4).
- To define a natural order using the `Comparable` interface (§14.5).
- To enable objects to listen for action events using the `ActionListener` interface (§14.6).
- To make objects cloneable using the `Cloneable` interface (§14.7).
- To explore the similarities and differences between an abstract class and an interface (§14.8).
- To create objects for primitive values using the wrapper classes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, and `Boolean`) (§14.9).
- To create a generic sort method (§14.10).
- To simplify programming using automatic conversion between primitive types and wrapper class types (§14.11).
- To use the `BigInteger` and `BigDecimal` classes for computing very large numbers with arbitrary precisions (§14.12).
- To design the `Rational` class for defining the `Rational` type (§14.13).



14.1 Introduction

problem

You have learned how to write simple programs to create and display GUI components. Can you write the code to respond to user actions, such as clicking a button? As shown in Figure 14.1, when a button is clicked, a message is displayed on the console.

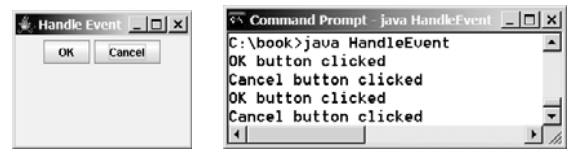


FIGURE 14.1 The program responds to button-clicking action events.

In order to write such code, you have to know interfaces. An interface is for defining common behavior for classes (especially unrelated classes). Before discussing interfaces, we introduce a closely related subject: abstract classes.

14.2 Abstract Classes

Video Note
Abstract **`GeometricObject` Class**

abstract class

In the inheritance hierarchy, classes become more specific and concrete *with each new subclass*. If you move from a subclass back up to a superclass, the classes become more general and less specific. Class design should ensure that a superclass contains common features of its subclasses. Sometimes a superclass is so abstract that it cannot have any specific instances. Such a class is referred to as an *abstract class*.

In Chapter 11, `GeometricObject` was defined as the superclass for `Circle` and `Rectangle`. `GeometricObject` models common features of geometric objects. Both `Circle` and `Rectangle` contain the `getArea()` and `getPerimeter()` methods for computing the area and perimeter of a circle and a rectangle. Since you can compute areas and perimeters for all geometric objects, it is better to define the `getArea()` and `getPerimeter()` methods in the `GeometricObject` class. However, these methods cannot be implemented in the `GeometricObject` class, because their implementation depends on the specific type of geometric object. Such methods are referred to as *abstract methods* and are denoted using the `abstract` modifier in the method header. After you define the methods in `GeometricObject`, it becomes an abstract class. Abstract classes are denoted using the `abstract` modifier in the class header. In UML graphic notation, the names of abstract classes and their abstract methods are italicized, as shown in Figure 14.2. Listing 14.1 gives the source code for the new `GeometricObject` class.

abstract method
`abstract` modifier

LISTING 14.1 GeometricObject.java

abstract class

```
1  public abstract class GeometricObject {
2    private String color = "white";
3    private boolean filled;
4    private java.util.Date dateCreated;
5
6    /** Construct a default geometric object */
7    protected GeometricObject() {
8      dateCreated = new java.util.Date();
9    }
10
11   /** Construct a geometric object with color and filled value */
12   protected GeometricObject(String color, boolean filled) {
13     dateCreated = new java.util.Date();
```

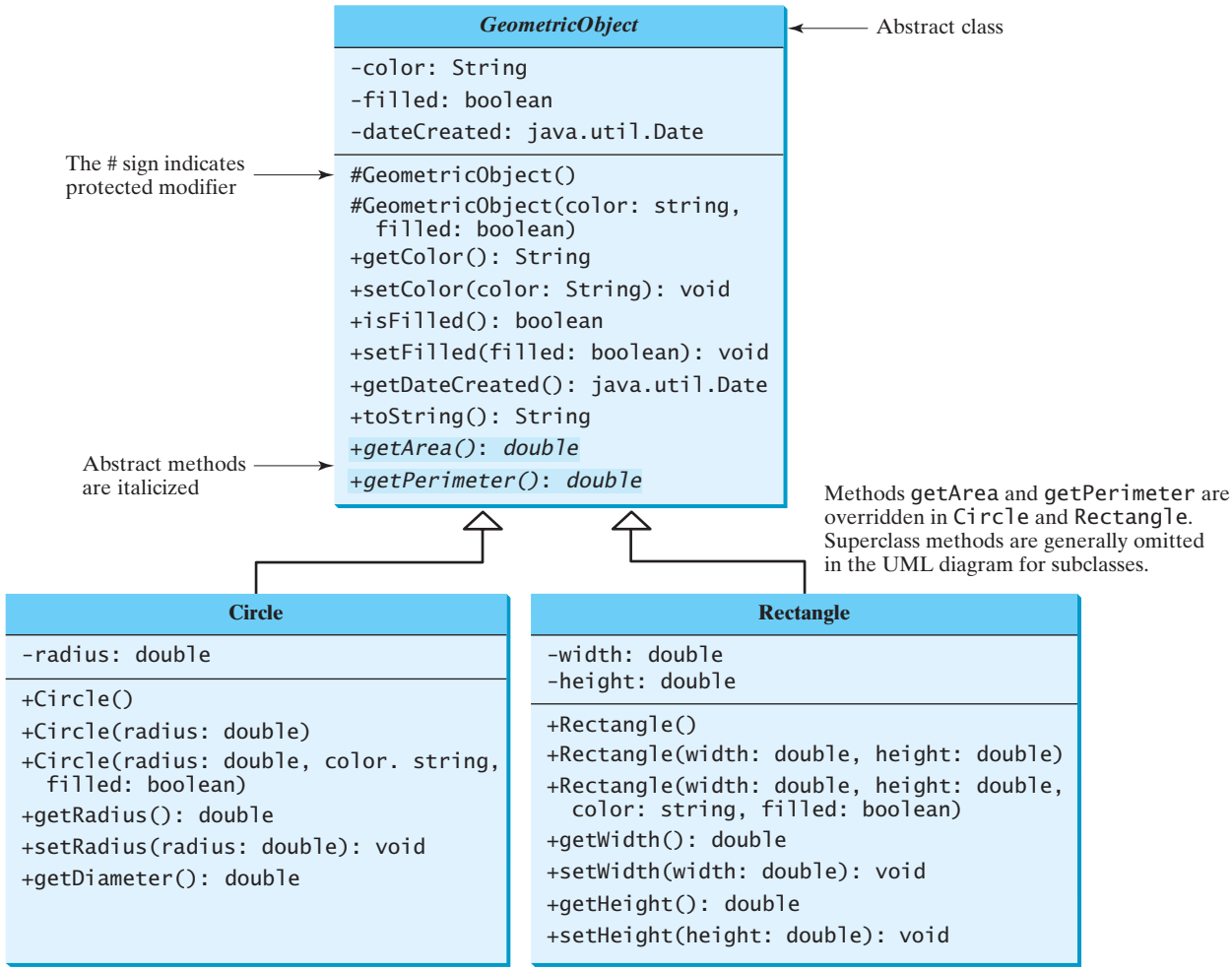


FIGURE 14.2 The new `GeometricObject` class contains abstract methods.

```
14      this.color = color;
15      this.filled = filled;
16    }
17
18    /** Return color */
19    public String getColor() {
20      return color;
21    }
22
23    /** Set a new color */
24    public void setColor(String color) {
25      this.color = color;
26    }
27
28    /** Return filled. Since filled is boolean,
29     *  the get method is named isFilled */
30    public boolean isFilled() {
31      return filled;
32    }
33
```

```
34   /** Set a new filled */
35   public void setFilled(boolean filled) {
36     this.filled = filled;
37   }
38
39   /** Get dateCreated */
40   public java.util.Date getDateCreated() {
41     return dateCreated;
42   }
43
44   /** Return a string representation of this object */
45   public String toString() {
46     return "created on " + dateCreated + "\ncolor: " + color +
47       " and filled: " + filled;
48   }
49
50   /** Abstract method getArea */
51   public abstract double getArea();
52
53   /** Abstract method getPerimeter */
54   public abstract double getPerimeter();
55 }
```

abstract method

abstract method

Abstract classes are like regular classes, but you cannot create instances of abstract classes using the `new` operator. An abstract method is defined without implementation. Its implementation is provided by the subclasses. A class that contains abstract methods must be defined abstract.

why protected constructor?

The constructor in the abstract class is defined protected, because it is used only by subclasses. When you create an instance of a concrete subclass, its superclass's constructor is invoked to initialize data fields defined in the superclass.

The `GeometricObject` abstract class defines the common features (data and methods) for geometric objects and provides appropriate constructors. Because you don't know how to compute areas and perimeters of geometric objects, `getArea` and `getPerimeter` are defined as abstract methods. These methods are implemented in the subclasses. The implementation of `Circle` and `Rectangle` is the same as in Listings 14.2 and 14.3, except that they extend the `GeometricObject` class defined in this chapter, as follows:

implementing **`Circle`**

implementing **`Rectangle`**

LISTING 14.2 Circle.java

extends abstract **`GeometricObject`**

```
1 public class Circle extends GeometricObject {
2   // Same as lines 2-46 in Listing 11.2, so omitted
3 }
```

LISTING 14.3 Rectangle.java

extends abstract **`GeometricObject`**

```
1 public class Rectangle extends GeometricObject {
2   // Same as lines 2-49 in Listing 11.3, so omitted
3 }
```

14.2.1 Why Abstract Methods?

You may be wondering what advantage is gained by defining the methods `getArea` and `getPerimeter` as abstract in the `GeometricObject` class instead of defining them only in each subclass. The following example shows the benefits of defining them in the `GeometricObject` class.

The example in Listing 14.4 creates two geometric objects, a circle and a rectangle, invokes the `equalArea` method to check whether they have equal areas and invokes the `displayGeometricObject` method to display them.

LISTING 14.4 TestGeometricObject.java

```
1 public class TestGeometricObject {
2   /** Main method */
3   public static void main(String[] args) {
4     // Create two geometric objects
5     GeometricObject geoObject1 = new Circle(5);                    create a circle
6     GeometricObject geoObject2 = new Rectangle(5, 3);              create a rectangle
7
8     System.out.println("The two objects have the same area? " +
9       equalArea(geoObject1, geoObject2) );
10
11    // Display circle
12    displayGeometricObject(geoObject1);
13
14    // Display rectangle
15    displayGeometricObject(geoObject2);
16  }
17
18  /** A method for comparing the areas of two geometric objects */
19  public static boolean equalArea(GeometricObject object1,          equalArea
20      GeometricObject object2) {
21    return object1.getArea() == object2.getArea();
22  }
23
24  /** A method for displaying a geometric object */
25  public static void displayGeometricObject(GeometricObject object) {   displayGeometricObject
26    System.out.println();
27    System.out.println("The area is " + object.getArea());
28    System.out.println("The perimeter is " + object.getPerimeter());
29  }
30 }
```

```
The two objects have the same area? false

The area is 78.53981633974483
The perimeter is 31.41592653589793

The area is 14.0
The perimeter is 16.0
```



The methods `getArea()` and `getPerimeter()` defined in the `GeometricObject` class are overridden in the `Circle` class and the `Rectangle` class. The statements (lines 5–6)

```
GeometricObject geoObject1 = new Circle(5);
GeometricObject geoObject2 = new Rectangle(5, 3);
```

create a new circle and rectangle and assign them to the variables `geoObject1` and `geoObject2`. These two variables are of the `GeometricObject` type.

When invoking `equalArea(geoObject1, geoObject2)` (line 9), the `getArea()` method defined in the `Circle` class is used for `object1.getArea()`, since `geoObject1` is a circle, and the `getArea()` method defined in the `Rectangle` class is used for `object2.getArea()`, since `geoObject2` is a rectangle.

Similarly, when invoking `displayGeometricObject(geoObject1)` (line 12), the methods `getArea` and `getPerimeter` defined in the `Circle` class are used, and when invoking `displayGeometricObject(geoObject2)` (line 15), the methods `getArea` and `getPerimeter` defined in the `Rectangle` class are used. The JVM dynamically determines which of these methods to invoke at runtime, depending on the type of object.

why abstract methods?

Note that you could not define the `equalArea` method for comparing whether two geometric objects have the same area if the `getArea` method were not defined in `GeometricObject`. So, you see the benefits of defining the abstract methods in `GeometricObject`.

14.2.2 Interesting Points on Abstract Classes

The following points on abstract classes are worth noting:

abstract method in abstract class

- An abstract method cannot be contained in a nonabstract class. If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined abstract. In other words, in a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented. Also note that abstract methods are nonstatic.

object cannot be created from abstract class

- An abstract class cannot be instantiated using the `new` operator, but you can still define its constructors, which are invoked in the constructors of its subclasses. For instance, the constructors of `GeometricObject` are invoked in the `Circle` class and the `Rectangle` class.

abstract class without abstract method

- A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that contains no abstract methods. In this case, you cannot create instances of the class using the `new` operator. This class is used as a base class for defining a new subclass.

superclass of abstract class may be concrete

- A subclass can be abstract even if its superclass is concrete. For example, the `Object` class is concrete, but its subclasses, such as `GeometricObject`, may be abstract.

concrete method overridden to be abstract

- A subclass can override a method from its superclass to define it `abstract`. This is *very unusual* but is useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be defined abstract.

abstract class as type

- You cannot create an instance from an abstract class using the `new` operator, but an abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of the `GeometricObject` type, is correct.

```
GeometricObject[] objects = new GeometricObject[10];
```

You can then create an instance of `GeometricObject` and assign its reference to the array like this:

```
objects[0] = new Circle();
```

14.3 Example: `Calendar` and `GregorianCalendar`



Video Note
`Calendar` and **`GregorianCalendar`** Classes

An instance of `java.util.Date` represents a specific instant in time with millisecond precision. `java.util.Calendar` is an abstract base class for extracting detailed calendar information, such as year, month, date, hour, minute, and second. Subclasses of `Calendar` can implement specific calendar systems, such as the Gregorian calendar, the lunar calendar, and the Jewish calendar. Currently, `java.util.GregorianCalendar` for the Gregorian calendar is supported in Java, as shown in Figure 14.3. The `add` method is abstract in the `Calendar` class, because its implementation is dependent on a concrete calendar system.

abstract **`add`** method

constructing calendar

You can use `new GregorianCalendar()` to construct a default `GregorianCalendar` with the current time and `new GregorianCalendar(year, month, date)` to construct a `GregorianCalendar` with the specified `year`, `month`, and `date`. The `month` parameter is `0` based—that is, `0` is for January.

`get(field)`

The `get(int field)` method defined in the `Calendar` class is useful for extracting the date and time information from a `Calendar` object. The fields are defined as constants, as shown in Table 14.1.

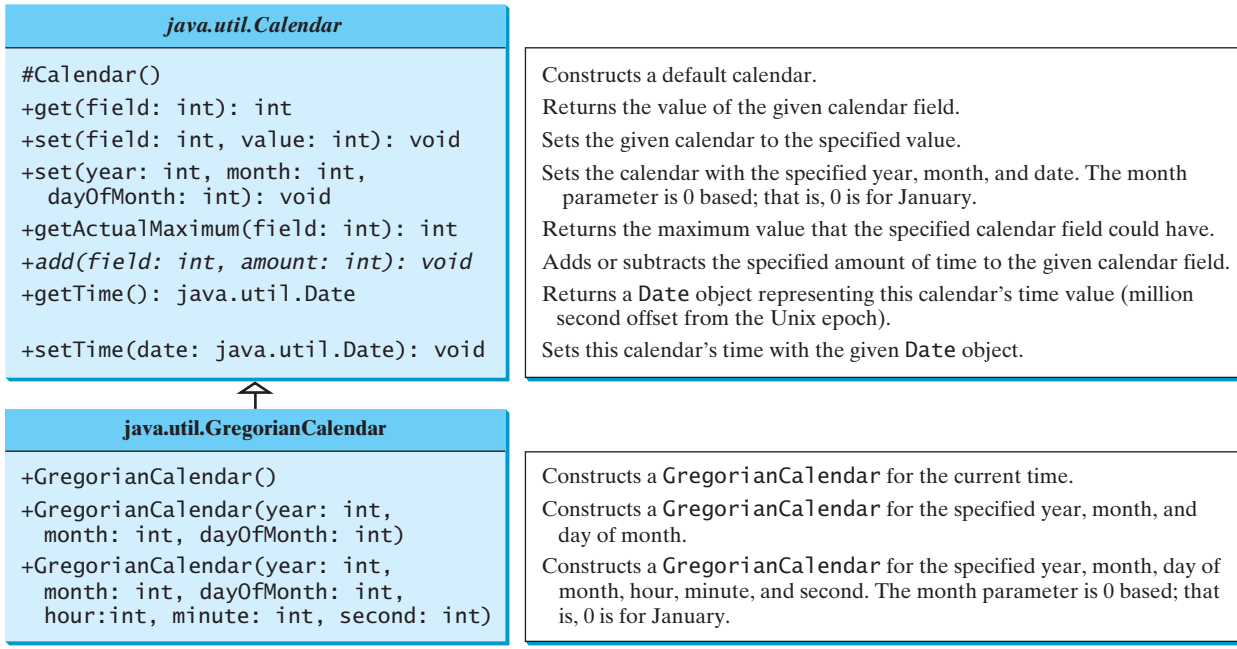

java.util.Calendar
#Calendar()
+get(field: int): int
+set(field: int, value: int): void
+set(year: int, month: int, dayOfMonth: int): void
+getActualMaximum(field: int): int
+add(field: int, amount: int): void
+getTime(): java.util.Date
+setTime(date: java.util.Date): void
Constructs a default calendar.
Returns the value of the given calendar field.
Sets the given calendar to the specified value.
Sets the calendar with the specified year, month, and date. The month parameter is 0 based; that is, 0 is for January.
Returns the maximum value that the specified calendar field could have.
Adds or subtracts the specified amount of time to the given calendar field.
Returns a Date object representing this calendar's time value (million second offset from the Unix epoch).
Sets this calendar's time with the given Date object.
java.util.GregorianCalendar
+GregorianCalendar()
+GregorianCalendar(year: int, month: int, dayOfMonth: int)
+GregorianCalendar(year: int, month: int, dayOfMonth: int, hour:int, minute: int, second: int)
Constructs a GregorianCalendar for the current time.
Constructs a GregorianCalendar for the specified year, month, and day of month.
Constructs a GregorianCalendar for the specified year, month, day of month, hour, minute, and second. The month parameter is 0 based; that is, 0 is for January.

FIGURE 14.3 The abstract `Calendar` class defines common features of various calendars.

TABLE 14.1 Field constants in the `Calendar` class

Constant	*Description*
`YEAR`	The year of the calendar.
`MONTH`	The month of the calendar with 0 for January.
`DATE`	The day of the calendar.
`HOUR`	The hour of the calendar (12-hour notation).
`HOUR_OF_DAY`	The hour of the calendar (24-hour notation).
`MINUTE`	The minute of the calendar.
`SECOND`	The second of the calendar.
`DAY_OF_WEEK`	The day number within the week with 1 for Sunday.
`DAY_OF_MONTH`	Same as DATE.
`DAY_OF_YEAR`	The day number in the year with 1 for the first day of the year.
`WEEK_OF_MONTH`	The week number within the month with 1 for the first week.
`WEEK_OF_YEAR`	The week number within the year with 1 for the first week.
`AM_PM`	Indicator for AM or PM (0 for AM and 1 for PM).

Listing 14.5 gives an example that displays the date and time information for the current time.

LISTING 14.5 TestCalendar.java

```
1 import java.util.*;
2
3 public class TestCalendar {
4   public static void main(String[] args) {
```

calendar for current time

extract fields in calendar

create a calendar

```
5     // Construct a Gregorian calendar for the current date and time
6     Calendar calendar = new GregorianCalendar();
7     System.out.println("Current time is " + new Date());
8     System.out.println("YEAR:\t" + calendar.get(Calendar.YEAR));
9     System.out.println("MONTH:\t" + calendar.get(Calendar.MONTH));
10    System.out.println("DATE:\t" + calendar.get(Calendar.DATE));
11    System.out.println("HOUR:\t" + calendar.get(Calendar.HOUR));
12    System.out.println("HOUR_OF_DAY:\t" +
13      calendar.get(Calendar.HOUR_OF_DAY));
14    System.out.println("MINUTE:\t" + calendar.get(Calendar.MINUTE));
15    System.out.println("SECOND:\t" + calendar.get(Calendar.SECOND));
16    System.out.println("DAY_OF_WEEK:\t" +
17      calendar.get(Calendar.DAY_OF_WEEK));
18    System.out.println("DAY_OF_MONTH:\t" +
19      calendar.get(Calendar.DAY_OF_MONTH));
20    System.out.println("DAY_OF_YEAR: " +
21      calendar.get(Calendar.DAY_OF_YEAR));
22    System.out.println("WEEK_OF_MONTH: " +
23      calendar.get(Calendar.WEEK_OF_MONTH));
24    System.out.println("WEEK_OF_YEAR: " +
25      calendar.get(Calendar.WEEK_OF_YEAR));
26    System.out.println("AM_PM: " + calendar.get(Calendar.AM_PM));
27
28    // Construct a calendar for September 11, 2001
29    Calendar calendar1 = new GregorianCalendar(2001, 8, 11);
30    System.out.println("September 11, 2001 is a " +
31      dayNameOfWeek(calendar1.get(Calendar.DAY_OF_WEEK)));
32  }
33
34  public static String dayNameOfWeek(int dayOfWeek) {
35    switch (dayOfWeek) {
36      case 1: return "Sunday";
37      case 2: return "Monday";
38      case 3: return "Tuesday";
39      case 4: return "Wednesday";
40      case 5: return "Thursday";
41      case 6: return "Friday";
42      case 7: return "Saturday";
43      default: return null;
44    }
45  }
46 }
```



```
Current time is Sun Sep 09 21:23:59 EDT 2007
YEAR:             2007
MONTH:            8
DATE:             9
HOUR:             9
HOUR_OF_DAY:      21
MINUTE:           23
SECOND:           59
DAY_OF_WEEK:      1
DAY_OF_MONTH:     9
DAY_OF_YEAR:      252
WEEK_OF_MONTH:    3
WEEK_OF_YEAR:     37
AM_PM:            1
September 11, 2001 is a Tuesday
```

The `set(int field, value)` method defined in the `Calendar` class can be used to set a field. For example, you can use `calendar.set(Calendar.DAY_OF_MONTH, 1)` to set the `calendar` to the first day of the month.

`set(field, value)`

The `add(field, value)` method adds the specified amount to a given field. For example, `add(Calendar.DAY_OF_MONTH, 5)` adds five days to the current time of the calendar. `add(Calendar.DAY_OF_MONTH, -5)` subtracts five days from the current time of the calendar.

`add(field, amount)`

To obtain the number of days in a month, use `calendar.getActualMaximum(Calendar.DAY_OF_MONTH)`. For example, if the `calendar` were for March, this method would return `31`.

`getActualMaximum(field)`

You can set a time represented in a `Date` object for the `calendar` by invoking `calendar. setTime(date)` and retrieve the time by invoking `calendar. getTime()`.

`setTime(Date)`
`getTime()`

14.4 Interfaces



Video Note
The concept of interface

An interface is a classlike construct that contains only constants and abstract methods. In many ways an interface is similar to an abstract class, but its intent is to specify common behavior for objects. For example, using appropriate interfaces, you can specify that the objects are comparable, edible, and/or cloneable.

To distinguish an interface from a class, Java uses the following syntax to define an interface:

```
modifier interface InterfaceName {
  /** Constant declarations */
  /** Method signatures */
}
```

Here is an example of an interface:

```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class. As with an abstract class, you cannot create an instance from an interface using the `new` operator, but in most cases you can use an interface more or less the same way you use an abstract class. For example, you can use an interface as a data type for a reference variable, as the result of casting, and so on.

You can now use the `Edible` interface to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the `implements` keyword. For example, the classes `Chicken` and `Fruit` in Listing 14.6 (lines 14, 23) implement the `Edible` interface. The relationship between the class and the interface is known as *interface inheritance*. Since interface inheritance and class inheritance are essentially the same, we will simply refer to both as inheritance.

interface inheritance

LISTING 14.6 TestEdible.java

```
1  public class TestEdible {
2    public static void main(String[] args) {
3      Object[] objects = {new Tiger(), new Chicken(), new Apple()};
4      for (int i = 0; i < objects.length; i++)
5        if (objects[i] instanceof Edible)
6          System.out.println(((Edible)objects[i]).howToEat());
7    }
8  }
9
10 class Animal {
```

`Animal` class

```
11   // Data fields, constructors, and methods omitted here
12 }
13
14 class Chicken extends Animal implements Edible {
15   public String howToEat() {
16     return "Chicken: Fry it";
17   }
18 }
19
20 class Tiger extends Animal {
21 }
22
23 abstract class Fruit implements Edible {
24   // Data fields, constructors, and methods omitted here
25 }
26
27 class Apple extends Fruit {
28   public String howToEat() {
29     return "Apple: Make apple cider";
30   }
31 }
32
33 class Orange extends Fruit {
34   public String howToEat() {
35     return "Orange: Make orange juice";
36   }
37 }
```

implements Edible

howToEat()

Tiger class

implements Edible

Apple class

Orange class



```
Chicken: Fry it
Apple: Make apple cider
```

The **Chicken** class extends **Animal** and implements **Edible** to specify that chickens are edible. When a class implements an interface, it implements all the methods defined in the interface with the exact signature and return type. The **Chicken** class implements the **howToEat** method (lines 15–17).

The **Fruit** class implements **Edible**. Since it does not implement the **howToEat** method, **Fruit** must be denoted **abstract** (line 23). The concrete subclasses of **Fruit** must implement the **howToEat** method. The **Apple** and **Orange** classes implements the **howToEat** method (lines 28, 34).

The **main** method creates an array with three objects for **Tiger**, **Chicken**, and **Apple** (line 3) and invokes the **howToEat** method if the element is edible (line 6).



Note

omitting modifiers

Since all data fields are ***public final static*** and all methods are ***public abstract*** in an interface, Java allows these modifiers to be omitted. Therefore the following interface definitions are equivalent:

```
public interface T {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T {
  int K = 1;

  void p();
}
```



Tip

A constant defined in an interface can be accessed using the syntax `InterfaceName.CONSTANT_NAME` (e.g., `T.K`). accessing constants

14.5 Example: The `Comparable` Interface

Suppose you want to design a generic method to find the larger of two objects of the same type, such as two students, two dates, two circles, two rectangles, or two squares. In order to accomplish this, the two objects must be comparable. So, the common behavior for the objects is comparable. Java provides the `Comparable` interface for this purpose. The interface is defined as follows:

`java.lang.Comparable`

```
// Interface for comparing objects, defined in java.lang
package java.lang;

public interface Comparable {
  public int compareTo(Object o);
}
```

The `compareTo` method determines the order of this object with the specified object `o` and returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than `o`.

Many classes in the Java library (e.g., `String` and `Date`) implement `Comparable` to define a natural order for the objects. If you examine the source code of these classes, you will see the keyword `implements` used in the classes, as shown below:

```
public class String extends Object
    implements Comparable {
  // class body omitted
}
```

```
public class Date extends Object
    implements Comparable {
  // class body omitted
}
```

Thus, strings are comparable, and so are dates. Let `s` be a `String` object and `d` be a `Date` object. All the following expressions are `true`.

```
s instanceof String
s instanceof Object
s instanceof Comparable
```

```
d instanceof java.util.Date
d instanceof Object
d instanceof Comparable
```

A generic `max` method for finding the larger of two objects can be defined, as shown in (a) or (b) below:

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum of two objects */
  public static Comparable max
      (Comparable o1, Comparable o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

(a)

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum of two objects */
  public static Object max
      (Object o1, Object o2) {
    if (((Comparable)o1).compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

(b)

(a) is simpler

The `max` method in (a) is simpler than the one in (b). In the `Max` class in (b), `o1` is declared as `Object`, and `(Comparable)o1` tells the compiler to cast `o1` into `Comparable` so that the `compareTo` method can be invoked from `o1`. However, no casting is needed in the `Max` class in (a), since `o1` is declared as `Comparable`.

(a) is more robust

The `max` method in (a) is more robust than the one in (b). You must invoke the `max` method with two comparable objects. Suppose you invoke `max` with two noncomparable objects:

```
Max.max(anyObject1, anyObject2);
```

The compiler will detect the error using the `max` method in (a), because `anyObject1` is not an instance of `Comparable`. Using the `max` method in (b), this line of code will compile fine but will have a runtime `ClassCastException`, because `anyObject1` is not an instance of `Comparable` and cannot be cast into `Comparable`.

From now on, assume that the `max` method in (a) is in the text. Since strings are comparable and dates are comparable, you can use the `max` method to find the larger of two instances of `String` or `Date`. Here is an example:

```
String s1 = "abcdef";
String s2 = "abcdee";
String s3 = (String)Max.max(s1, s2);
```

```
Date d1 = new Date();
Date d2 = new Date();
Date d3 = (Date)Max.max(d1, d2);
```

The `return` value from the `max` method is of the `Comparable` type. So you need to cast it to `String` or `Date` explicitly.

You cannot use the `max` method to find the larger of two instances of `Rectangle`, because `Rectangle` does not implement `Comparable`. However, you can define a new rectangle class that implements `Comparable`. The instances of this new class are comparable. Let this new class be named `ComparableRectangle`, as shown in Listing 14.7.

LISTING 14.7 ComparableRectangle.java

implements **Comparable**

implement **compareTo**

```
1  public class ComparableRectangle extends Rectangle
2      implements Comparable {
3    /** Construct a ComparableRectangle with specified properties */
4    public ComparableRectangle(double width, double height) {
5      super(width, height);
6    }
7
8    /** Implement the compareTo method defined in Comparable */
9    public int compareTo(Object o) {
10     if (getArea() > ((ComparableRectangle)o).getArea())
11       return 1;
12     else if (getArea() < ((ComparableRectangle)o).getArea())
13       return -1;
14     else
15       return 0;
16   }
17 }
```

`ComparableRectangle` extends `Rectangle` and implements `Comparable`, as shown in Figure 14.4. The keyword `implements` indicates that `ComparableRectangle` inherits all the constants from the `Comparable` interface and implements the methods in the interface. The `compareTo` method compares the areas of two rectangles. An instance of `ComparableRectangle` is also an instance of `Rectangle`, `GeometricObject`, `Object`, and `Comparable`.

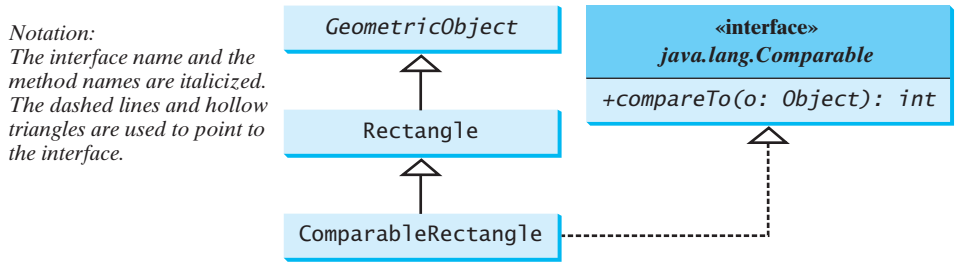


FIGURE 14.4 `ComparableRectangle` extends `Rectangle` and implements `Comparable`.

You can now use the `max` method to find the larger of two objects of `ComparableRectangle`. Here is an example:

```
ComparableRectangle rectangle1 = new ComparableRectangle(4, 5);
ComparableRectangle rectangle2 = new ComparableRectangle(3, 6);
System.out.println(Max.max(rectangle1, rectangle2));
```

benefits of interface

An interface provides another form of generic programming. It would be difficult to use a generic `max` method to find the maximum of the objects without using an interface in this example, because multiple inheritance would be necessary to inherit `Comparable` and another class, such as `Rectangle`, at the same time.

The `Object` class contains the `equals` method, which is intended for the subclasses of the `Object` class to override in order to compare whether the contents of the objects are the same. Suppose that the `Object` class contains the `compareTo` method, as defined in the `Comparable` interface; the new `max` method can be used to compare a list of *any* objects. Whether a `compareTo` method should be included in the `Object` class is debatable. Since the `compareTo` method is not defined in the `Object` class, the `Comparable` interface is defined in Java to enable objects to be compared if they are instances of the `Comparable` interface. It is strongly recommended (though not required) that `compareTo` should be consistent with `equals`. That is, for two objects `o1` and `o2`, `o1.compareTo(o2) == 0` if and only if `o1.equals(o2)` is `true`.

14.6 Example: The `ActionListener` Interface

Now you are ready to write a short program to address the problem proposed in the introduction of this chapter. The program displays two buttons in the frame, as shown in Figure 14.1. To respond to a button click, you need to write the code to process the button-clicking action. The button is a *source object* where the action originates. You need to create an object capable of handling the action event on a button. This object is called a *listener*, as shown in Figure 14.5.

Not all objects can be listeners for an action event. To be a listener, two requirements must be met:

`ActionListener` interface

1. The object must be an instance of the `ActionListener` interface. This interface defines the common behavior for all action listeners.

`addActionListener(listener)`

2. The `ActionListener` object `listener` must be registered with the source using the method `source.addActionListener(listener)`.

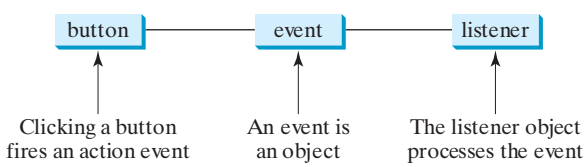


FIGURE 14.5 A listener object processes the event fired from the source object.

The `ActionListener` interface contains the `actionPerformed` method for processing the event. Your listener class must override this method to respond to the event. Listing 14.8 gives the code that processes the `ActionEvent` on the two buttons. When you click the *OK* button, the message "OK button clicked" is displayed. When you click the *Cancel* button, the message "Cancel button clicked" is displayed, as shown in Figure 14.1.

LISTING 14.8 HandleEvent.java

```
1  import javax.swing.*;
2  import java.awt.event.*;
3
4  public class HandleEvent extends JFrame {
5    public HandleEvent() {
6      // Create two buttons
7      JButton jbtOK = new JButton("OK");
8      JButton jbtCancel = new JButton("Cancel");
9
10     // Create a panel to hold buttons
11     JPanel panel = new JPanel();
12     panel.add(jbtOK);
13     panel.add(jbtCancel);
14
15     add(panel); // Add panel to the frame
16
17     // Register listeners
18     OKListenerClass listener1 = new OKListenerClass();
19     CancelListenerClass listener2 = new CancelListenerClass();
20     jbtOK.addActionListener(listener1);
21     jbtCancel.addActionListener(listener2);
22   }
23
24   public static void main(String[] args) {
25     JFrame frame = new HandleEvent();
26     frame.setTitle("Handle Event");
27     frame.setSize(200, 150);
28     frame.setLocation(200, 100);
29     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30     frame.setVisible(true);
31   }
32 }
33
34 class OKListenerClass implements ActionListener {
35   public void actionPerformed(ActionEvent e) {
36     System.out.println("OK button clicked");
37   }
38 }
39
40 class CancelListenerClass implements ActionListener {
41   public void actionPerformed(ActionEvent e) {
42     System.out.println("Cancel button clicked");
43   }
44 }
```

create listener

register listener

listener class process event

listener class process event

Two listener classes are defined in lines 34–44. Each listener class implements `ActionListener` to process `ActionEvent`. The object `listener1` is an instance of `OKListenerClass` (line 18), which is registered with the button `jbtOK` (line 20). When the *OK* button is clicked, the `actionPerformed(ActionEvent)` method (line 36) in `OKListenerClass` is invoked to process the event. The object `listener2` is an instance of `CancelListenerClass` (line 19), which is registered with the button `jbtCancel` in line 21. When the *OK* button is clicked, the

actionPerformed(ActionEvent) method (line 42) in CancelListenerClass is invoked to process the event. Ways to process events will be discussed further in Chapter 16, "Event-Driven Programming."

14.7 Example: The Cloneable Interface

Often it is desirable to create a copy of an object. You need to use the clone method and understand the Cloneable interface, which is the subject of this section.

An interface contains constants and abstract methods, but the Cloneable interface is a special case. The Cloneable interface in the java.lang package is defined as follows:

java.lang.Cloneable

```
package java.lang;

public interface Cloneable {
}
```

This interface is empty. An interface with an empty body is referred to as a *marker interface*. A marker interface does not contain constants or methods. It is used to denote that a class possesses certain desirable properties. A class that implements the Cloneable interface is marked cloneable, and its objects can be cloned using the clone() method defined in the Object class.

marker interface

Many classes in the Java library (e.g., Date, Calendar, and ArrayList) implement Cloneable. Thus, the instances of these classes can be cloned. For example, the following code

```
1 Calendar calendar = new GregorianCalendar(2003, 2, 1);
2 Calendar calendar1 = calendar;
3 Calendar calendar2 = (Calendar)calendar.clone();
4 System.out.println("calendar == calendar1 is " +
5   (calendar == calendar1));
6 System.out.println("calendar == calendar2 is " +
7   (calendar == calendar2));
8 System.out.println("calendar.equals(calendar2) is " +
9   calendar.equals(calendar2));
```

displays

```
calendar == calendar1 is true
calendar == calendar2 is false
calendar.equals(calendar2) is true
```

In the preceding code, line 2 copies the reference of calendar to calendar1, so calendar and calendar1 point to the same Calendar object. Line 3 creates a new object that is the clone of calendar and assigns the new object's reference to calendar2. calendar2 and calendar are different objects with the same contents.

You can clone an array using the clone method. For example, the following code

clone arrays

```
1 int[] list1 = {1, 2};
2 int[] list2 = list1.clone();
3 list1[0] = 7; list1[1] = 8;
4
5 System.out.println("list1 is " + list1[0] + ", " + list1[1]);
6 System.out.println("list2 is " + list2[0] + ", " + list2[1]);
```

displays

```
list1 is 7, 8
list2 is 1, 2
```

how to implement **Cloneable**

To define a custom class that implements the **Cloneable** interface, the class must override the **clone()** method in the **Object** class. Listing 14.9 defines a class named **House** that implements **Cloneable** and **Comparable**.

LISTING 14.9 House.java

```
1  public class House implements Cloneable, Comparable {
2    private int id;
3    private double area;
4    private java.util.Date whenBuilt;
5
6    public House(int id, double area) {
7      this.id = id;
8      this.area = area;
9      whenBuilt = new java.util.Date();
10   }
11
12   public int getId() {
13     return id;
14   }
15
16   public double getArea() {
17     return area;
18   }
19
20   public java.util.Date getWhenBuilt() {
21     return whenBuilt;
22   }
23
24   /** Override the protected clone method defined in the Object
25     class, and strengthen its accessibility */
26   public Object clone() throws CloneNotSupportedException {
27     return super.clone();
28   }
29
30   /** Implement the compareTo method defined in Comparable */
31   public int compareTo(Object o) {
32     if (area > ((House)o).area)
33       return 1;
34     else if (area < ((House)o).area)
35       return -1;
36     else
37       return 0;
38   }
39 }
```

This exception is thrown if **House** does not implement **Cloneable**

The **House** class implements the **clone** method (lines 26–28) defined in the **Object** class. The header is:

```
protected native Object clone() throws CloneNotSupportedException;
```

The keyword **native** indicates that this method is not written in Java but is implemented in the JVM for the native platform. The keyword **protected** restricts the method to be accessed in the same package or in a subclass. For this reason, the **House** class must override the method and change the visibility modifier to **public** so that the method can be used in any package. Since the **clone** method implemented for the native platform in the **Object** class performs the task of cloning objects, the **clone** method in the **House** class simply invokes **super.clone()**. The **clone** method defined in the **Object** class may throw **CloneNotSupportedException**.

CloneNotSupported-Exception

The `House` class implements the `compareTo` method (lines 31–38) defined in the `Comparable` interface. The method compares the areas of two houses.

You can now create an object of the `House` class and create an identical copy from it, as follows:

```
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
```

`house1` and `house2` are two different objects with identical contents. The `clone` method in the `Object` class copies each field from the original object to the target object. If the field is of a primitive type, its value is copied. For example, the value of `area` (`double` type) is copied from `house1` to `house2`. If the field is of an object, the reference of the field is copied. For example, the field `whenBuilt` is of the `Date` class, so its reference is copied into `house2`, as shown in Figure 14.6. Therefore, `house1.whenBuilt == house2.whenBuilt` is true, although `house1 == house2` is false. This is referred to as a *shallow copy* rather than a *deep copy*, meaning that if the field is of an object type, the object's reference is copied rather than its contents.

shallow copy

deep copy

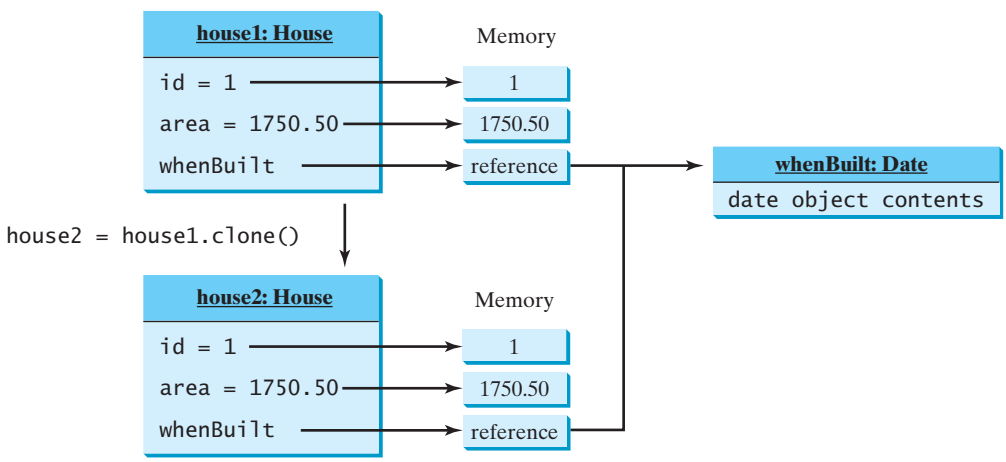


FIGURE 14.6 The `clone` method copies the values of primitive type fields and the references of object type fields.

If you want to perform a deep copy, you can override the `clone` method with custom cloning operations after invoking `super.clone()`. See Exercise 14.4.



Caution

If the `House` class does not override the `clone()` method, the program will receive a syntax error, because `clone()` is protected in `java.lang.Object`. If `House` does not implement `java.lang.Cloneable`, invoking `super.clone()` (line 27) in House.java will cause a `CloneNotSupportedException`. Thus, to enable cloning an object, the class for the object must override the `clone()` method and implement `Cloneable`.

14.8 Interfaces vs. Abstract Classes

An interface can be used more or less the same way as an abstract class, but defining an interface is different from defining an abstract class. Table 14.2 summarizes the differences.

Java allows only single inheritance for class extension but allows multiple extensions for interfaces. For example,

TABLE 14.2 Interfaces vs. Abstract Classes

	Variables	*Constructors*	*Methods*
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be `public static final`.	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods.

```
public class NewClass extends BaseClass
    implements Interface1, ..., InterfaceN {
  ...
}
```

subinterface

An interface can inherit other interfaces using the `extends` keyword. Such an interface is called a *subinterface*. For example, `NewInterface` in the following code is a subinterface of `Interface1`, . . . , and `InterfaceN`.

```
public interface NewInterface extends Interface1, ..., InterfaceN {
  // constants and abstract methods
}
```

A class implementing `NewInterface` must implement the abstract methods defined in `NewInterface`, `Interface1`, . . . , and `InterfaceN`. An interface can extend other interfaces but not classes. A class can extend its superclass and implement multiple interfaces.

All classes share a single root, the `Object` class, but there is no single root for interfaces. Like a class, an interface also defines a type. A variable of an interface type can reference any instance of the class that implements the interface. If a class implements an interface, the interface is like a superclass for the class. You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa. For example, suppose that `c` is an instance of `Class2` in Figure 14.7. `c` is also an instance of `Object`, `Class1`, `Interface1`, `Interface1_1`, `Interface1_2`, `Interface2_1`, and `Interface2_2`.

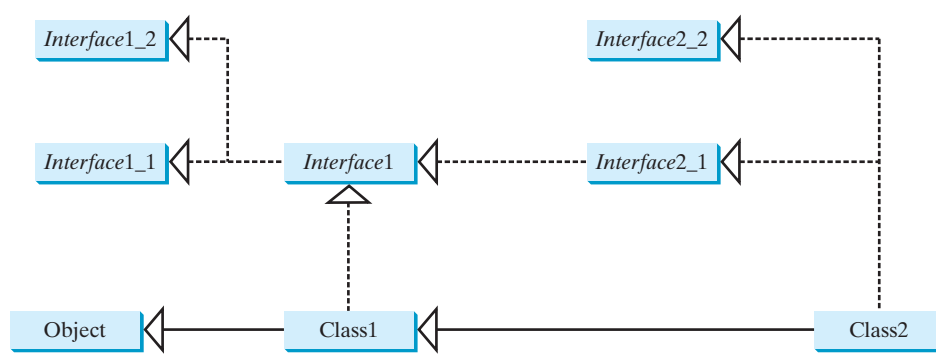


FIGURE 14.7 `Class1` implements `Interface1`; `Interface1` extends `Interface1_1` and `Interface1_2`. `Class2` extends `Class1` and implements `Interface2_1` and `Interface2_2`.



Note

Class names are nouns. Interface names may be adjectives or nouns. For example, both `java.lang.Comparable` and `java.awt.event.ActionListener` are interfaces. `Comparable` is an adjective, and `ActionListener` is a noun.

naming convention



Design Guide

Abstract classes and interfaces can both be used to specify common behavior of objects. How do you decide whether to use an interface or a class? In general, a *strong is-a relationship* that clearly describes a parent-child relationship should be modeled using classes. For example, Gregorian calendar is a calendar, so the relationship between the class `java.util.GregorianCalendar` and `java.util.Calendar` is modeled using class inheritance. A *weak is-a relationship*, also known as an *is-kind-of relationship*, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces. For example, all strings are comparable, so the `String` class implements the `Comparable` interface.

is-a relationship
is-kind-of relationship

In general, interfaces are preferred over abstract classes because an interface can define a common supertype for unrelated classes. Interfaces are more flexible than classes. Consider the `Animal` class. Suppose the `howToEat` method is defined in the `Animal` class, as follows:

interface preferred

```
abstract class Animal {
  public abstract String howToEat();
}
```

Animal class

Two subclasses of `Animal` are defined as follows:

```
class Chicken extends Animal {
  public String howToEat() {
    return "Fry it";
  }
}
```

Chicken class

```
class Duck extends Animal {
  public String howToEat() {
    return "Roast it";
  }
}
```

Duck class

Given this inheritance hierarchy, polymorphism enables you to hold a reference to a `Chicken` object or a `Duck` object in a variable of type `Animal`, as in the following code:

```
public static void main(String[] args) {
  Animal animal = new Chicken();
  eat(animal);

  animal = new Duck();
  eat(animal);
}

public static void eat(Animal animal) {
  animal.howToEat();
}
```

The JVM dynamically decides which `howToEat` method to invoke based on the actual object that invokes the method.

You can define a subclass of `Animal`. However, there is a restriction. The subclass must be for another animal (e.g., `Turkey`).

Interfaces don't have this restriction. Interfaces give you more flexibility than classes, because you don't have make everything fit into one type of class. You may define the

howToEat() method in an interface and let it serve as a common supertype for other classes. For example,

```
public static void main(String[] args) {
  Edible stuff = new Chicken();
  eat(stuff);

  stuff = new Duck();
  eat(stuff);

  stuff = new Broccoli();
  eat(stuff);
}

public static void eat(Edible stuff) {
  stuff.howToEat();
}
```

Edible interface

```
interface Edible {
  public String howToEat();
}
```

Chicken class

```
class Chicken implements Edible {
  public String howToEat() {
    return "Fry it";
  }
}
```

Duck class

```
class Duck implements Edible {
  public String howToEat() {
    return "Roast it";
  }
}
```

Broccoli class

```
class Broccoli implements Edible {
  public String howToEat() {
    return "Stir-fry it";
  }
}
```

To define a class that represents edible objects, simply let the class implement the Edible interface. The class is now a subtype of the Edible type. Any Edible object can be passed to invoke the eat method.

14.9 Processing Primitive Data Type Values as Objects

Owing to performance considerations, primitive data types are not used as objects in Java. Because of the overhead of processing objects, the language's performance would be adversely affected if primitive data types were treated as objects. However, many Java methods require the use of objects as arguments. For example, the add(object) method in the ArrayList class adds an object to an ArrayList. Java offers a convenient way to incorporate, or wrap, a primitive data type into an object (e.g., wrapping int into the Integer class, and wrapping double into the Double class). The corresponding class is called a *wrapper class*. By using a wrapper object instead of a primitive data type variable, you can take advantage of generic programming.

why wrapper class?

Java provides Boolean, Character, Double, Float, Byte, Short, Integer, and Long wrapper classes for primitive data types. These classes are grouped in the java.lang package. Their inheritance hierarchy is shown in Figure 14.8.

why wrapper class?

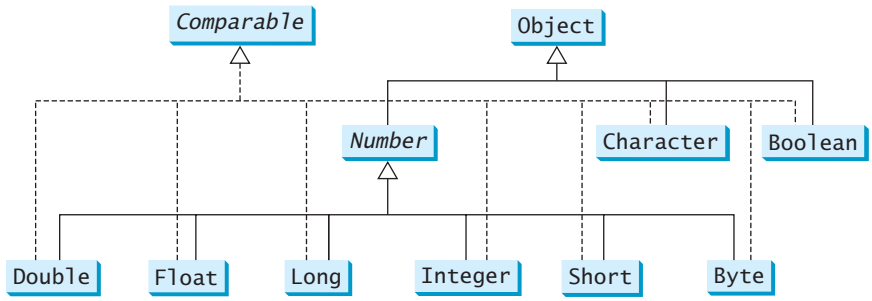


FIGURE 14.8 The `Number` class is an abstract superclass for `Double`, `Float`, `Long`, `Integer`, `Short`, and `Byte`.



Note

Most wrapper class names for a primitive type are the same as the primitive data type name with the first letter capitalized. The exceptions are `Integer` and `Character`.

naming convention

Each numeric wrapper class extends the abstract `Number` class, which contains the methods `doubleValue()`, `floatValue()`, `intValue()`, `longValue()`, `shortValue()`, and `byteValue()`. These methods "convert" objects into primitive type values. Each wrapper class overrides the `toString` and `equals` methods defined in the `Object` class. Since all the wrapper classes implement the `Comparable` interface, the `compareTo` method is implemented in these classes.

Wrapper classes are very similar to each other. The `Character` class was introduced in Chapter 9, "Strings and Text I/O." The `Boolean` class wraps a Boolean value `true` or `false`. This section uses `Integer` and `Double` as examples to introduce the numeric wrapper classes. The key features of `Integer` and `Double` are shown in Figure 14.9.

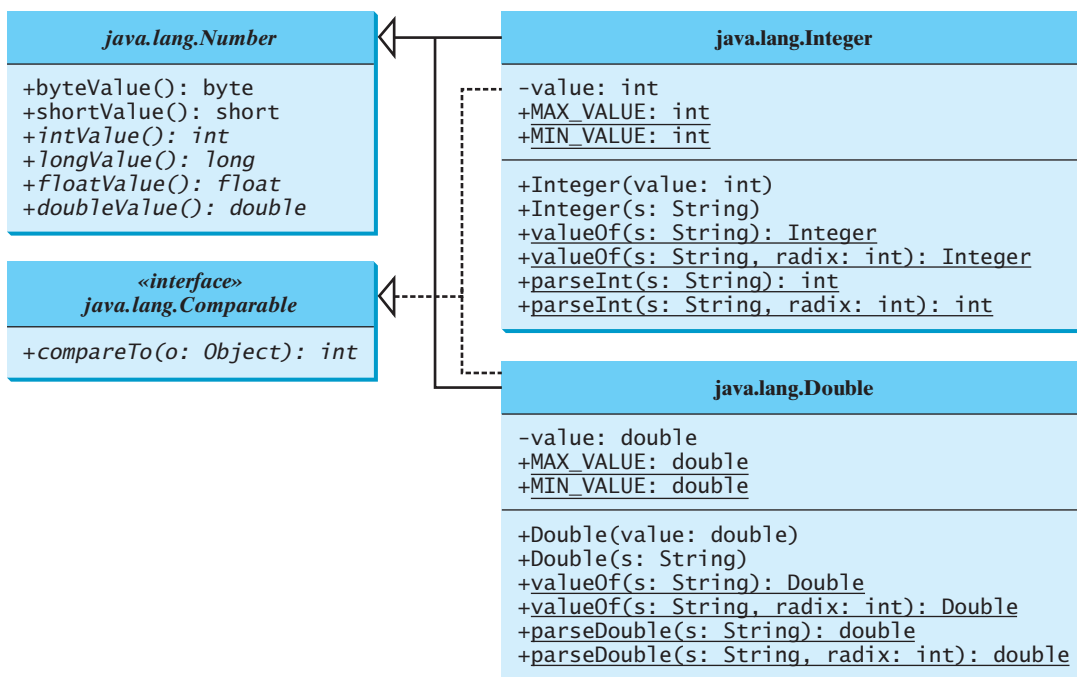


FIGURE 14.9 The wrapper classes provide constructors, constants, and conversion methods for manipulating various data types.

constructors

You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value—for example, `new Double(5.0)`, `new Double("5.0")`, `new Integer(5)`, and `new Integer("5")`.

no no-arg constructor

The wrapper classes do not have no-arg constructors.

immutable

The instances of all wrapper classes are immutable; this means that, once the objects are created, their internal values cannot be changed.

constants

Each numeric wrapper class has the constants `MAX_VALUE` and `MIN_VALUE`. `MAX_VALUE` represents the maximum value of the corresponding primitive data type. For `Byte`, `Short`, `Integer`, and `Long`, `MIN_VALUE` represents the minimum `byte`, `short`, `int`, and `long` values. For `Float` and `Double`, `MIN_VALUE` represents the minimum *positive* `float` and `double` values. The following statements display the maximum integer (2,147,483,647), the minimum positive float (1.4E−45), and the maximum double floating-point number (1.79769313486231570e+308d).

```
System.out.println("The maximum integer is " + Integer.MAX_VALUE);
System.out.println("The minimum positive float is " +
  Float.MIN_VALUE);
System.out.println(
  "The maximum double-precision floating-point number is " +
  Double.MAX_VALUE);
```

conversion methods

Each numeric wrapper class implements the abstract methods `doubleValue()`, `floatValue()`, `intValue()`, `longValue()`, and `shortValue()`, which are defined in the `Number` class. These methods return a `double`, `float`, `int`, `long`, or `short` value for the wrapper object.

static `valueOf` methods

The numeric wrapper classes have a useful static method, `valueOf(String s)`. This method creates a new object initialized to the value represented by the specified string. For example,

```
Double doubleObject = Double.valueOf("12.4");
Integer integerObject = Integer.valueOf("12");
```

static parsing methods

You have used the `parseInt` method in the `Integer` class to parse a numeric string into an `int` value and the `parseDouble` method in the `Double` class to parse a numeric string into a `double` value. Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value based on `10` (decimal) or any specified radix (e.g., `2` for binary, `8` for octal, and `16` for hexadecimal). These methods are shown below:

```
// These two methods are in the Byte class
public static byte parseByte(String s)
public static byte parseByte(String s, int radix)

// These two methods are in the Short class
public static short parseShort(String s)
public static short parseShort(String s, int radix)

// These two methods are in the Integer class
public static int parseInt(String s)
public static int parseInt(String s, int radix)

// These two methods are in the Long class
public static long parseLong(String s)
public static long parseLong(String s, int radix)

// These two methods are in the Float class
public static float parseFloat(String s)
public static float parseFloat(String s, int radix)
```

```
// These two methods are in the Double class
public static double parseDouble(String s)
public static double parseDouble(String s, int radix)
```

For example,

```
Integer.parseInt("11", 2) returns 3;
Integer.parseInt("12", 8) returns 10;
Integer.parseInt("13", 10) returns 13;
Integer.parseInt("1A", 16) returns 26;
```

`Integer.parseInt("12", 2)` would raise a runtime exception because `12` is not a binary number.

14.10 Sorting an Array of Objects

This example presents a static generic method for sorting an array of comparable objects. The objects are instances of the `Comparable` interface, and they are compared using the `compareTo` method. The method can be used to sort an array of any objects as long as their classes implement the `Comparable` interface.

To test the method, the program sorts an array of integers, an array of double numbers, an array of characters, and an array of strings. The program is shown in Listing 14.10.

LISTING 14.10 GenericSort.java

```
1  public class GenericSort {
2    public static void main(String[] args) {
3      // Create an Integer array
4      Integer[] intArray = {new Integer(2), new Integer(4),
5        new Integer(3)};
6
7      // Create a Double array
8      Double[] doubleArray = {new Double(3.4), new Double(1.3),
9        new Double(-22.1)};
10
11     // Create a Character array
12     Character[] charArray = {new Character('a'),
13       new Character('J'), new Character('r')};
14
15     // Create a String array
16     String[] stringArray = {"Tom", "John", "Fred"};
17
18     // Sort the arrays
19     sort(intArray);
20     sort(doubleArray);
21     sort(charArray);
22     sort(stringArray);
23
24     // Display the sorted arrays
25     System.out.print("Sorted Integer objects: ");
26     printList(intArray);
27     System.out.print("Sorted Double objects: ");
28     printList(doubleArray);
29     System.out.print("Sorted Character objects: ");
30     printList(charArray);
31     System.out.print("Sorted String objects: ");
32     printList(stringArray);
33   }
34
```

sort **Integer** objects
sort **Double** objects
sort **Character** objects
sort **String** objects

generic sort method

```
35   /** Sort an array of comparable objects */
36   public static void sort(Comparable[] list) {
37     Comparable currentMin;
38     int currentMinIndex;
39
40     for (int i = 0; i < list.length - 1; i++) {
41       // Find the maximum in the list[0..i]
42       currentMin = list[i];
43       currentMinIndex = i;
44
45       for (int j = i + 1; j < list.length; j++) {
46         if (currentMin.compareTo(list[j]) > 0) {
47           currentMin = list[j];
48           currentMinIndex = j;
49         }
50       }
51
52       // Swap list[i] with list[currentMinIndex] if necessary;
53       if (currentMinIndex != i) {
54         list[currentMinIndex] = list[i];
55         list[i] = currentMin;
56       }
57     }
58   }
59
60   /** Print an array of objects */
61   public static void printList(Object[] list) {
62     for (int i = 0; i < list.length; i++)
63       System.out.print(list[i] + " ");
64     System.out.println();
65   }
66 }
```

compareTo



```
Sorted Integer objects: 2 3 4
Sorted Double objects: -22.1 1.3 3.4
Sorted Character objects: J a r
Sorted String objects: Fred John Tom
```

The algorithm for the `sort` method is the same as in §6.10.1, "Selection Sort." The `sort` method in §6.10.1 sorts an array of `double` values. The `sort` method in this example can sort an array of any object type, provided that the objects are also instances of the `Comparable` interface. This is another example of *generic programming*. Generic programming enables a method to operate on arguments of generic types, making it reusable with multiple types.

`Integer`, `Double`, `Character`, and `String` implement `Comparable`, so the objects of these classes can be compared using the `compareTo` method. The `sort` method uses the `compareTo` method to determine the order of the objects in the array.



Tip

Arrays.sort method

Java provides a static `sort` method for sorting an array of any object type in the `java.util.Arrays` class, provided that the elements in the array are comparable. Thus you can use the following code to sort arrays in this example:

```
java.util.Arrays.sort(intArray);
java.util.Arrays.sort(doubleArray);
java.util.Arrays.sort(charArray);
java.util.Arrays.sort(stringArray);
```



Note

Arrays are objects. An array is an instance of the `Object` class. Furthermore, if `A` is a subtype of `B`, every instance of `A[]` is an instance of `B[]`. Therefore, the following statements are all true:

```
new int[10] instanceof Object
new Integer[10] instanceof Object
new Integer[10] instanceof Comparable[]
new Integer[10] instanceof Number[]
new Number[10] instanceof Object[]
```

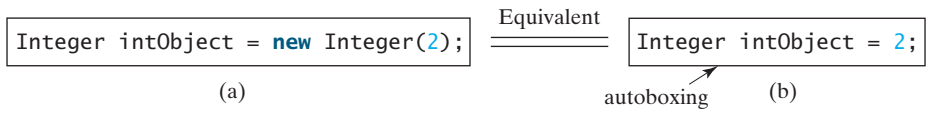


Caution

Although an `int` value can be assigned to a `double` type variable, `int[]` and `double[]` are two incompatible types. Therefore, you cannot assign an `int[]` array to a variable of `double[]` or `Object[]` type.

14.11 Automatic Conversion between Primitive Types and Wrapper Class Types

Java allows primitive types and wrapper classes to be converted automatically. For example, the following statement in (a) can be simplified as in (b) due to autoboxing:



boxing
unboxing

Converting a primitive value to a wrapper object is called *boxing*. The reverse conversion is called *unboxing*. The compiler will automatically box a primitive value that appears in a context requiring an object, and will unbox an object that appears in a context requiring a primitive value. Consider the following example:

```
1 Integer[] intArray = {1, 2, 3};
2 System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

In line 1, primitive values `1`, `2`, and `3` are automatically boxed into objects `new Integer(1)`, `new Integer(2)`, and `new Integer(3)`. In line 2, objects `intArray[0]`, `intArray[1]`, and `intArray[2]` are automatically converted into `int` values that are added together.

14.12 The `BigInteger` and `BigDecimal` Classes

immutable

If you need to compute with very large integers or high-precision floating-point values, you can use the `BigInteger` and `BigDecimal` classes in the `java.math` package. Both are *immutable*. Both extend the `Number` class and implement the `Comparable` interface. The largest integer of the `long` type is `Long.MAX_VALUE` (i.e., `9223372036854775807`). An instance of `BigInteger` can represent an integer of any size. You can use `new BigInteger(String)` and `new BigDecimal(String)` to create an instance of `BigInteger` and `BigDecimal`, use the `add`, `subtract`, `multiple`, `divide`, and `remainder` methods to perform arithmetic operations, and use the `compareTo` method to compare two big numbers. For example, the following code creates two `BigInteger` objects and multiplies them.

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```

The output is `18446744073709551614`.

There is no limit to the precision of a `BigDecimal` object. The `divide` method may throw an `ArithmeticException` if the result cannot be terminated. However, you can use the overloaded `divide(BigDecimal d, int scale, int roundingMode)` method to specify a scale and a rounding mode to avoid this exception, where `scale` is the minimum number of digits after the decimal point. For example, the following code creates two `BigDecimal` objects and performs division with scale `20` and rounding mode `BigDecimal.ROUND_UP`.

```
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
```

The output is `0.33333333333333333334`.

Note that the factorial of an integer can be very large. Listing 14.11 gives a method that can return the factorial of any integer.

LISTING 14.11 LargeFactorial.java

```
1 import java.math.*;
2
3 public class LargeFactorial {
4   public static void main(String[] args) {
5     System.out.println("50! is \n" + factorial(50));
6   }
7
8   public static BigInteger factorial(long n) {
9     BigInteger result = BigInteger.ONE;
10    for (int i = 1; i <= n; i++)
11      result = result.multiply(new BigInteger(i + ""));
12
13    return result;
14  }
15 }
```

constant

multiply



```
50! is
30414093201713378043612608166064768844377641568960512000000000000
```

`BigInteger.ONE` (line 9) is a constant defined in the `BigInteger` class. `BigInteger.ONE` is same as `new BigInteger("1")`.

A new result is obtained by invoking the `multiply` method (line 11).

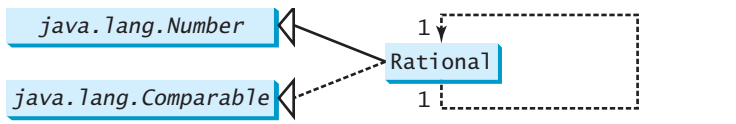
14.13 Case Study: The `Rational` Class

A rational number has a numerator and a denominator in the form `a/b`, where `a` is the numerator and `b` the denominator. For example, `1/3`, `3/4`, and `10/4` are rational numbers.

A rational number cannot have a denominator of `0`, but a numerator of `0` is fine. Every integer `i` is equivalent to a rational number `i/1`. Rational numbers are used in exact computations involving fractions—for example, `1/3` = `0.33333` . . . . This number cannot be precisely represented in floating-point format using data type `double` or `float`. To obtain the exact result, we must use rational numbers.

Java provides data types for integers and floating-point numbers, but not for rational numbers. This section shows how to design a class to represent rational numbers.

Since rational numbers share many common features with integers and floating-point numbers, and **Number** is the root class for numeric wrapper classes, it is appropriate to define **Rational** as a subclass of **Number**. Since rational numbers are comparable, the **Rational** class should also implement the **Comparable** interface. Figure 14.10 illustrates the **Rational** class and its relationship to the **Number** class and the **Comparable** interface.



Rational	
-numerator: long -denominator: long	The numerator of this rational number. The denominator of this rational number.
+Rational()	Creates a rational number with numerator 0 and denominator 1.
+Rational(numerator: long, denominator: long)	Creates a rational number with specified numerator and denominator.
+getNumerator(): long	Returns the numerator of this rational number.
+getDenominator(): long	Returns the denominator of this rational number.
+add(secondRational: Rational): Rational	Returns the addition of this rational with another.
+subtract(secondRational: Rational): Rational	Returns the subtraction of this rational with another.
+multiply(secondRational: Rational): Rational	Returns the multiplication of this rational with another.
+divide(secondRational: Rational): Rational	Returns the division of this rational with another.
+toString(): String	Returns a string in the form "numerator / denominator." Returns numerator if denominator is 1.
-gcd(n: long, d: long): long	Returns the greatest common divisor between n and d.

FIGURE 14.10 The properties, constructors, and methods of the **Rational** class are illustrated in UML.

A rational number consists of a numerator and a denominator. There are many equivalent rational numbers—for example, **1/3 = 2/6 = 3/9 = 4/12**. The numerator and the denominator of **1/3** have no common divisor except **1**, so **1/3** is said to be in lowest terms.

To reduce a rational number to its lowest terms, you need to find the greatest common divisor (GCD) of the absolute values of its numerator and denominator, then divide both numerator and denominator by this value. You can use the method for computing the GCD of two integers **n** and **d**, as suggested in Listing 4.8, GreatestCommonDivisor.java. The numerator and denominator in a **Rational** object are reduced to their lowest terms.

As usual, let us first write a test program to create two **Rational** objects and test its methods. Listing 14.12 is a test program.

LISTING 14.12 TestRationalClass.java

```
1 public class TestRationalClass {
2   /** Main method */
3   public static void main(String[] args) {
4     // Create and initialize two rational numbers r1 and r2.
5     Rational r1 = new Rational(4, 2);          create a Rational
6     Rational r2 = new Rational(2, 3);          create a Rational
7
```

add

```
 8     // Display results
 9     System.out.println(r1 + " + " + r2 + " = " + r1.add(r2));
10     System.out.println(r1 + " - " + r2 + " = " + r1.subtract(r2));
11     System.out.println(r1 + " * " + r2 + " = " + r1.multiply(r2));
12     System.out.println(r1 + " / " + r2 + " = " + r1.divide(r2));
13     System.out.println(r2 + " is " + r2.doubleValue());
14   }
15 }
```



```
2 + 2/3 = 8/3
2 - 2/3 = 4/3
2 * 2/3 = 4/3
2 / 2/3 = 3
2/3 is 0.6666666666666666
```

The `main` method creates two rational numbers, `r1` and `r2` (lines 5–6), and displays the results of `r1 + r2`, `r1 - r2`, `r1 x r2`, and `r1 / r2` (lines 9–12). To perform `r1 + r2`, invoke `r1.add(r2)` to return a new `Rational` object. Similarly, `r1.subtract(r2)` is for `r1 - r2`, `r1.multiply(r2)` for `r1 x r2`, and `r1.divide(r2)` for `r1 / r2`.

The `doubleValue()` method displays the double value of `r2` (line 13). The `doubleValue()` method is defined in `java.lang.Number` and overridden in `Rational`.

Note that when a string is concatenated with an object using the plus sign (+), the object's string representation from the `toString()` method is used to concatenate with the string. So `r1 + " + " + r2 + " = " + r1.add(r2)` is equivalent to `r1.toString() + " + " + r2.toString() + " = " + r1.add(r2).toString()`.

The `Rational` class is implemented in Listing 14.13.

LISTING 14.13 Rational.java

```
1  public class Rational extends Number implements Comparable {
2    // Data fields for numerator and denominator
3    private long numerator = 0;
4    private long denominator = 1;
5
6    /** Construct a rational with default properties */
7    public Rational() {
8      this(0, 1);
9    }
10
11   /** Construct a rational with specified numerator and denominator */
12   public Rational(long numerator, long denominator) {
13     long gcd = gcd(numerator, denominator);
14     this.numerator = ((denominator > 0) ? 1 : -1) * numerator / gcd;
15     this.denominator = Math.abs(denominator) / gcd;
16   }
17
18   /** Find GCD of two numbers */
19   private static long gcd(long n, long d) {
20     long n1 = Math.abs(n);
21     long n2 = Math.abs(d);
22     int gcd = 1;
23
24     for (int k = 1; k <= n1 && k <= n2; k++) {
25       if (n1 % k == 0 && n2 % k == 0)
26         gcd = k;
27     }
```

```
28
29      return gcd;
30    }
31
32    /** Return numerator */
33    public long getNumerator() {
34      return numerator;
35    }
36
37    /** Return denominator */
38    public long getDenominator() {
39      return denominator;
40    }
41
42    /** Add a rational number to this rational */
43    public Rational add(Rational secondRational) {
44      long n = numerator * secondRational.getDenominator() +
45        denominator * secondRational.getNumerator();
46      long d = denominator * secondRational.getDenominator();
47      return new Rational(n, d);
48    }
49
50    /** Subtract a rational number from this rational */
51    public Rational subtract(Rational secondRational) {
52      long n = numerator * secondRational.getDenominator()
53        - denominator * secondRational.getNumerator();
54      long d = denominator * secondRational.getDenominator();
55      return new Rational(n, d);
56    }
57
58    /** Multiply a rational number to this rational */
59    public Rational multiply(Rational secondRational) {
60      long n = numerator * secondRational.getNumerator();
61      long d = denominator * secondRational.getDenominator();
62      return new Rational(n, d);
63    }
64
65    /** Divide a rational number from this rational */
66    public Rational divide(Rational secondRational) {
67      long n = numerator * secondRational.getDenominator();
68      long d = denominator * secondRational.numerator;
69      return new Rational(n, d);
70    }
71
72    /** Override the toString() method */
73    public String toString() {
74      if (denominator == 1)
75        return numerator + "";
76      else
77        return numerator + "/" + denominator;
78    }
79
80    /** Override the equals method in the Object class */
81    public boolean equals(Object parm1) {
82      if ((this.subtract((Rational)(parm1))).getNumerator() == 0)
83        return true;
84      else
85        return false;
86    }
87
```

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

$$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$$

$$\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$$

```
88    /** Implement the abstract intValue method in java.lang.Number */
89    public int intValue() {
90      return (int)doubleValue();
91    }
92
93    /** Implement the abstract floatValue method in java.lang.Number */
94    public float floatValue() {
95      return (float)doubleValue();
96    }
97
98    /** Implement the doubleValue method in java.lang.Number */
99    public double doubleValue() {
100     return numerator * 1.0 / denominator;
101   }
102
103   /** Implement the abstract longValue method in java.lang.Number */
104   public long longValue() {
105     return (long)doubleValue();
106   }
107
108   /** Implement the compareTo method in java.lang.Comparable */
109   public int compareTo(Object o) {
110     if ((this.subtract((Rational)o)).getNumerator() > 0)
111       return 1;
112     else if ((this.subtract((Rational)o)).getNumerator() < 0)
113       return -1;
114     else
115       return 0;
116   }
117 }
```

The rational number is encapsulated in a `Rational` object. Internally, a rational number is represented in its lowest terms (line 13), and the numerator determines its sign (line 14). The denominator is always positive (line 15).

The `gcd()` method (lines 19–30 in the `Rational` class) is private; it is not intended for use by clients. The `gcd()` method is only for internal use by the `Rational` class. The `gcd()` method is also static, since it is not dependent on any particular `Rational` object.

The `abs(x)` method (lines 20–21 in the `Rational` class) is defined in the `Math` class that returns the absolute value of `x`.

Two `Rational` objects can interact with each other to perform add, subtract, multiply, and divide operations. These methods return a new `Rational` object (lines 43–70).

The methods `toString` and `equals` in the `Object` class are overridden in the `Rational` class (lines 73–91). The `toString()` method returns a string representation of a `Rational` object in the form `numerator/denominator`, or simply `numerator` if `denominator` is `1`. The `equals(Object other)` method returns true if this rational number is equal to the other rational number.

The abstract methods `intValue`, `longValue`, `floatValue`, and `doubleValue` in the `Number` class are implemented in the `Rational` class (lines 88–106). These methods return `int`, `long`, `float`, and `double` value for this rational number.

The `compareTo(Object other)` method in the `Comparable` interface is implemented in the `Rational` class (lines 109–116) to compare this rational number to the other rational number.



Tip

The `get` methods for the properties `numerator` and `denominator` are provided in the `Rational` class, but the `set` methods are not provided, so, once a `Rational` object is created,

its contents cannot be changed. The `Rational` class is immutable. The `String` class and the wrapper classes for primitive type values are also immutable.

immutable



Tip

The numerator and denominator are represented using two variables. It is possible to use an array of two integers to represent the numerator and denominator. See Exercise 14.18. The signatures of the public methods in the `Rational` class are not changed, although the internal representation of a rational number is changed. This is a good example to illustrate the idea that the data fields of a class should be kept private so as to encapsulate the implementation of the class from the use of the class.

encapsulation

The `Rational` class has serious limitations. It can easily overflow. For example, the following code will display an incorrect result, because the denominator is too large.

overflow

```
public class Test {
  public static void main(String[] args) {
    Rational r1 = new Rational(1, 123456789);
    Rational r2 = new Rational(1, 123456789);
    Rational r3 = new Rational(1, 123456789);
    System.out.println("r1 * r2 * r3 is " +
      r1.multiply(r2.multiply(r3)));
  }
}
```

```
r1 * r2 * r3 is -1/2204193661661244627
```



To fix it, you may implement the `Rational` class using the `BigInteger` for numerator and denominator (see Exercise 14.19).

Key Terms

abstract class 458
abstract method 458
deep copy 473
interface 465
marker interface 471
multiple inheritance 469
subinterface 474
shallow copy 473
single inheritance 473
wrapper class 476

Chapter Summary

1. Abstract classes are like regular classes with data and methods, but you cannot create instances of abstract classes using the `new` operator.
2. An abstract method cannot be contained in a nonabstract class. If a subclass of an abstract superclass does not implement all the inherited abstract methods of the superclass, the subclass must be defined abstract.
3. A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that contains no abstract methods.
4. A subclass can be abstract even if its superclass is concrete.
5. An interface is a classlike construct that contains only constants and abstract methods. In many ways, an interface is similar to an abstract class, but an abstract class can contain constants and abstract methods as well as variables and concrete methods.

6. An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class.

7. The `java.lang.Comparable` interface defines the `compareTo` method. Many classes in the Java library implement `Comparable`.

8. The `java.lang.Cloneable` interface is a marker interface. An object of the class that implements the `Cloneable` interface is cloneable.

9. A class can extend only one superclass but can implement one or more interfaces.

10. An interface can extend one or more interfaces.

11. Many Java methods require the use of objects as arguments. Java offers a convenient way to incorporate, or wrap, a primitive data type into an object (e.g., wrapping `int` into the `Integer` class, and wrapping `double` into the `Double` class). The corresponding class is called a *wrapper class*. By using a wrapper object instead of a primitive data type variable, you can take advantage of generic programming.

12. Java can automatically convert a primitive type value to its corresponding wrapper object in the context and vice versa.

13. The `BigInteger` class is useful to compute and process integers of any size. The `BigDecimal` class can be used to compute and process floating-point numbers with any arbitrary precision.

Review Questions

Section 14.2

14.1 Which of the following classes definitions defines a legal abstract class?

```
class A {
  abstract void unfinished() {
  }
}
```

(a)

```
public class abstract A {
  abstract void unfinished();
}
```

(b)

```
class A {
  abstract void unfinished();
}
```

(c)

```
abstract class A {
  protected void unfinished();
}
```

(d)

```
abstract class A {
  abstract void unfinished();
}
```

(e)

```
abstract class A {
  abstract int unfinished();
}
```

(f)

14.2 The `getArea` and `getPerimeter` methods may be removed from the `GeometricObject` class. What are the benefits of defining `getArea` and `getPerimeter` as abstract methods in the `GeometricObject` class?

14.3 True or false? An abstract class can be used just like a nonabstract class except that you cannot use the `new` operator to create an instance from the abstract class.

Sections 14.4–14.6

14.4 Which of the following is a correct interface?

```
interface A {
  void print() { };
}
```

(a)

```
abstract interface A extends I1, I2 {
  abstract void print() { };
}
```

(b)

```
abstract interface A {
  print();
}
```

(c)

```
interface A {
  void print();
}
```

(d)

14.5 True or false? If a class implements `Comparable`, the object of the class can invoke the `compareTo` method.

14.6 Two `max` methods are defined in §14.5. Explain why the `max` with the signature `max(Comparable, Comparable)` is better than the one with the signature `max(Object, Object)`.

14.7 You can define the `compareTo` method in a class without implementing the `Comparable` interface. What are the benefits of implementing the `Comparable` interface?

14.8 True or false? If a class implements `java.awt.event.ActionListener`, the object of the class can invoke the `actionPerformed` method.

Sections 14.7–14.8

14.9 Can you invoke the `clone()` method to clone an object if the class for the object does not implement the `java.lang.Cloneable`? Does the `Date` class implement `Cloneable`?

14.10 What would happen if the `House` class (defined in Listing 14.9) did not override the `clone()` method or if `House` did not implement `java.lang.Cloneable`?

14.11 Show the printout of the following code:

```
java.util.Date date = new java.util.Date();
java.util.Date date1 = date;
java.util.Date date2 = (java.util.Date)(date.clone());
System.out.println(date == date1);
System.out.println(date == date2);
System.out.println(date.equals(date2));
```

14.12 Show the printout of the following code:

```
java.util.ArrayList list = new java.util.ArrayList();
list.add("New York");
java.util.ArrayList list1 = list;
java.util.ArrayList list2 = (java.util.ArrayList)(list.clone());
list.add("Atlanta");
System.out.println(list == list1);
System.out.println(list == list2);
System.out.println("list is " + list);
System.out.println("list1 is " + list1);
System.out.println("list2.get(0) is " + list2.get(0));
System.out.println("list2.size() is " + list2.size());
```

14.13 What is wrong in the following code?

```
public class Test {
  public static void main(String[] args) {
    GeometricObject x = new Circle(3);
    GeometricObject y = x.clone();
    System.out.println(x == y);
  }
}
```

14.14 Give an example to show why interfaces are preferred over abstract classes.

Section 14.9

14.15 Describe primitive-type wrapper classes. Why do you need these wrapper classes?

14.16 Can each of the following statements be compiled?

```
Integer i = new Integer("23");
Integer i = new Integer(23);
Integer i = Integer.valueOf("23");
Integer i = Integer.parseInt("23", 8);
Double d = new Double();
Double d = Double.valueOf("23.45");
int i = (Integer.valueOf("23")).intValue();
double d = (Double.valueOf("23.4")).doubleValue();
int i = (Double.valueOf("23.4")).intValue();
String s = (Double.valueOf("23.4")).toString();
```

14.17 How do you convert an integer into a string? How do you convert a numeric string into an integer? How do you convert a double number into a string? How do you convert a numeric string into a double value?

14.18 Why do the following two lines of code compile but cause a runtime error?

```
Number numberRef = new Integer(0);
Double doubleRef = (Double)numberRef;
```

14.19 Why do the following two lines of code compile but cause a runtime error?

```
Number[] numberArray = new Integer[2];
numberArray[0] = new Double(1.5);
```

14.20 What is wrong in the following code?

```
public class Test {
  public static void main(String[] args) {
    Number x = new Integer(3);
    System.out.println(x.intValue());
    System.out.println(x.compareTo(new Integer(4)));
  }
}
```

14.21 What is wrong in the following code?

```
public class Test {
  public static void main(String[] args) {
    Number x = new Integer(3);
    System.out.println(x.intValue());
    System.out.println((Integer)x.compareTo(new Integer(4)));
  }
}
```

14.22 What is the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    System.out.println(Integer.parseInt("10"));
    System.out.println(Integer.parseInt("10", 10));
    System.out.println(Integer.parseInt("10", 16));
    System.out.println(Integer.parseInt("11"));
    System.out.println(Integer.parseInt("11", 10));
    System.out.println(Integer.parseInt("11", 16));
  }
}
```

Sections 14.10–14.12

14.23 What are autoboxing and autounboxing? Are the following statements correct?

```
Number x = 3;
Integer x = 3;
Double x = 3;
Double x = 3.0;
int x = new Integer(3);
int x = new Integer(3) + new Integer(4);
double y = 3.4;
y.intValue();

JOptionPane.showMessageDialog(null, 45.5);
```

14.24 Can you assign `new int[10]`, `new String[100]`, `new Object[50]`, or `new Calendar[20]` into a variable of `Object[]` type?

14.25 What is the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    java.math.BigInteger x = new java.math.BigInteger("3");
    java.math.BigInteger y = new java.math.BigInteger("7");
    x.add(y);
    System.out.println(x);
  }
}
```

Comprehensive

14.26 Define the following terms: abstract classes, interfaces. What are the similarities and differences between abstract classes and interfaces?

14.27 Indicate true or false for the following statements:

- An abstract class can have instances created using the constructor of the abstract class.
- An abstract class can be extended.
- An interface is compiled into a separate bytecode file.
- A subclass of a nonabstract superclass cannot be abstract.
- A subclass cannot override a concrete method in a superclass to define it abstract.
- An abstract method must be nonstatic
- An interface can have static methods.
- An interface can extend one or more interfaces.

- An interface can extend an abstract class.
- An abstract class can extend an interface.

Programming Exercises

Sections 14.1–14.7

14.1* (*Enabling `GeometricObject` comparable*) Modify the `GeometricObject` class to implement the `Comparable` interface, and define a static `max` method in the `GeometricObject` class for finding the larger of two `GeometricObject` objects. Draw the UML diagram and implement the new `GeometricObject` class. Write a test program that uses the `max` method to find the larger of two circles and the larger of two rectangles.

14.2* (*The `ComparableCircle` class*) Create a class named `ComparableCircle` that extends `Circle` and implements `Comparable`. Draw the UML diagram and implement the `compareTo` method to compare the circles on the basis of area. Write a test class to find the larger of two instances of `ComparableCircle` objects.

14.3* (*The `Colorable` interface*) Design an interface named `Colorable` with a `void` method named `howToColor()`. Every class of a colorable object must implement the `Colorable` interface. Design a class named `Square` that extends `GeometricObject` and implements `Colorable`. Implement `howToColor` to display a message `"Color all four sides"`.

Draw a UML diagram that involves `Colorable`, `Square`, and `GeometricObject`. Write a test program that creates an array of five `GeometricObject`s. For each object in the array, invoke its `howToColor` method if it is colorable.

14.4* (*Revising the `House` class*) Rewrite the `House` class in Listing 14.9 to perform a deep copy on the `whenBuilt` field.

14.5* (*Enabling `Circle` comparable*) Rewrite the `Circle` class in Listing 14.2 to extend `GeometricObject` and implement the `Comparable` interface. Override the `equals` method in the `Object` class. Two `Circle` objects are equal if their radii are the same. Draw the UML diagram that involves `Circle`, `GeometricObject`, and `Comparable`.



Video Note
Redesign the `Rectangle` class

14.6* (*Enabling `Rectangle` comparable*) Rewrite the `Rectangle` class in Listing 14.3 to extend `GeometricObject` and implement the `Comparable` interface. Override the `equals` method in the `Object` class. Two `Rectangle` objects are equal if their areas are the same. Draw the UML diagram that involves `Rectangle`, `GeometricObject`, and `Comparable`.

14.7* (*The `Octagon` class*) Write a class named `Octagon` that extends `GeometricObject` and implements the `Comparable` and `Cloneable` interfaces. Assume that all eight sides of the octagon are of equal size. The area can be computed using the following formula:

$$area = \left(2 + 4/\sqrt{2}\right) * side * side$$

Draw the UML diagram that involves `Octagon`, `GeometricObject`, `Comparable`, and `Cloneable`. Write a test program that creates an `Octagon` object with side value `5` and displays its area and perimeter. Create a new object using the `clone` method and compare the two objects using the `compareTo` method.

14.8* (*Summing the areas of geometric objects*) Write a method that sums the areas of all the geometric objects in an array. The method signature is:

```
public static double sumArea(GeometricObject[] a)
```

Write a test program that creates an array of four objects (two circles and two rectangles) and computes their total area using the `sumArea` method.

14.9* (*Finding the largest object*) Write a method that returns the largest object in an array of objects. The method signature is:

```
public static Object max(Comparable[] a)
```

All the objects are instances of the `Comparable` interface. The order of the objects in the array is determined using the `compareTo` method.

Write a test program that creates an array of ten strings, an array of ten integers, and an array of ten dates, and finds the largest string, integer, and date in the arrays.

14.10** (*Displaying calendars*) Rewrite the `PrintCalendar` class in Listing 5.12 to display a calendar for a specified month using the `Calendar` and `GregorianCalendar` classes. Your program receives the month and year from the command line. For example:

```
java Exercise14_10 1 2010
```

This displays the calendar shown in Figure 14.11.

You also can run the program without the year. In this case, the year is the current year. If you run the program without specifying a month and a year, the month is the current month.

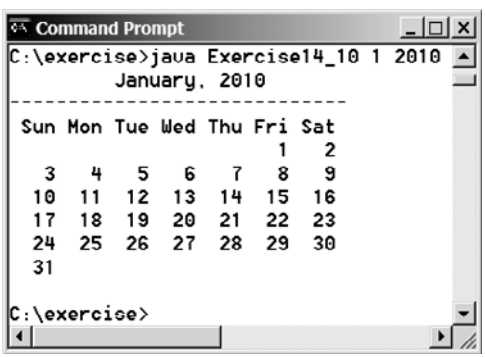

Command Prompt
C:\exercise>java Exercise14_10 1 2010
January, 2010
Sun Mon Tue Wed Thu Fri Sat
1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
C:\exercise>

FIGURE 14.11 The program displays a calendar for January 2010.

Section 14.12

14.11** (*Divisible by 5 or 6*) Find the first ten numbers (greater than `Long.MAX_VALUE`) that are divisible by `5` or `6`.

14.12** (*Divisible by 2 or 3*) Find the first ten numbers with `50` decimal digits that are divisible by `2` or `3`.

14.13** (*Square numbers*) Find the first ten square numbers that are greater than `Long.MAX_VALUE`. A square number is a number in the form of n^2.

14.14** (*Large prime numbers*) Write a program that finds five prime numbers larger than `Long.MAX_VALUE`.

14.15** (*Mersenne prime*) A prime number is called a *Mersenne prime* if it can be written in the form $2^p - 1$ for some positive integer p. Write a program that finds all Mersenne primes with $p \leq 100$ and displays the output as shown below. (You have to use `BigInteger` to store the number, because it is too big to be stored in `long`. Your program may take several hours to complete.)

```
p         2^p - 1

2           3
3           7
5           31
...
```

Section 14.13

14.16** (*Approximating e*) Exercise 4.26 approximates e using the following series:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots + \frac{1}{i!}$$

In order to get better precision, use `BigDecimal` with `25` digits of precision in the computation. Write a program that displays the `e` value for `i` = `100`, `200`, ..., and `1000`.

14.17 (*Using the `Rational` class*) Write a program that will compute the following summation series using the `Rational` class:

$$\frac{1}{2} + \frac{2}{3} + \frac{3}{4} + \ldots + \frac{98}{99} + \frac{99}{100}$$

You will discover that output is incorrect because of integer overflow (too large). To fix this problem, see Exercise 14.19.

14.18* (*Demonstrating the benefits of encapsulation*) Rewrite the `Rational` class in §14.13 using a new internal representation for numerator and denominator. Create an array of two integers as follows:

```
private long[] r = new long[2];
```

Use `r[0]` to represent the numerator and `r[1]` to represent the denominator. The signatures of the methods in the `Rational` class are not changed, so a client application that uses the previous `Rational` class can continue to use this new `Rational` class without being recompiled.

14.19** (*Using `BigInteger` for the `Rational` class*) Redesign and implement the `Rational` class in §14.13 using `BigInteger` for numerator and denominator.

14.20* (*Creating a rational-number calculator*) Write a program similar to Listing 9.5, Calculator.java. Instead of using integers, use rationals, as shown in Figure 14.12.

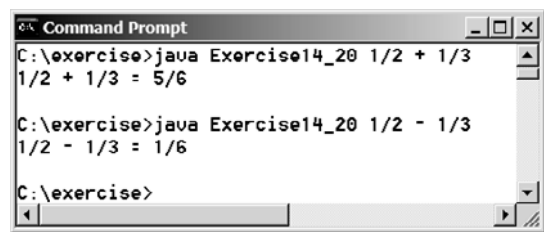


FIGURE 14.12 The program takes three arguments (operand1, operator, and operand2) from the command line and displays the expression and the result of the arithmetic operation.

You will need to use the `split` method in the `String` class, introduced in §9.2.6, "Converting, Replacing, and Splitting Strings," to retrieve the numerator string and denominator string, and convert strings into integers using the `Integer.parseInt` method.

14.21* (*Math: The `Complex` class*) A complex number is a number of the form $a + bi$, where a and b are real numbers and i is $\sqrt{-1}$. The numbers `a` and `b` are known as the real part and imaginary part of the complex number, respectively. You can perform addition, subtraction, multiplication, and division for complex numbers using the following formula:

$$a + bi + c + di = (a + c) + (b + d)i$$
$$a + bi - (c + di) = (a - c) + (b - d)i$$
$$(a + bi)*(c + di) = (ac - bd) + (bc + ad)i$$
$$(a + bi)/(c + di) = (ac + bd)/(c^2 + d^2) + (bc - ad)i/(c^2 + d^2)$$

You can also obtain the absolute value for a complex number using the following formula:

$$|a + bi| = \sqrt{a^2 + b^2}$$

Design a class named `Complex` for representing complex numbers and the methods `add`, `subtract`, `multiply`, `divide`, and `abs` for performing complex-number operations, and override the `toString` method for returning a string representation for a complex number. The `toString` method returns `a + bi` as a string. If `b` is `0`, it simply returns `a`.

Provide three constructors `Complex(a, b)`, `Complex(a)`, and `Complex()`. `Complex()` creates a `Complex` object for number `0` and `Complex(a)` creates a `Complex` object with `0` for `b`. Also provide the `getRealPart()` and `getImaginaryPart()` methods for returning the real and imaginary part of the complex number, respectively.

Write a test program that prompts the user to enter two complex numbers and display the result of their addition, subtraction, multiplication, and division. Here is a sample run:



```
Enter the first complex number: 3.5 5.5 ↵Enter
Enter the second complex number: -3.5 1 ↵Enter
3.5 + 5.5i + -3.5 + 1.0i = 0.0 + 6.5i
3.5 + 5.5i - -3.5 + 1.0i = 7.0 + 4.5i
3.5 + 5.5i * -3.5 + 1.0i = -17.75 + -15.75i
3.5 + 5.5i / -3.5 + 1.0i = -0.5094 + -1.7i
|3.5 + 5.5i| = 6.519202405202649
```

This page intentionally left blank

CHAPTER 15

GRAPHICS

Objectives

- To describe Java coordinate systems in a GUI component (§15.2).
- To draw things using the methods in the `Graphics` class (§15.3).
- To override the `paintComponent` method to draw things on a GUI component (§15.3).
- To use a panel as a canvas to draw things (§15.3).
- To draw strings, lines, rectangles, ovals, arcs, and polygons (§§15.4, 15.6–15.7).
- To obtain font properties using `FontMetrics` and know how to center a message (§15.8).
- To display an image in a GUI component (§15.11).
- To develop reusable GUI components `FigurePanel`, `MessagePanel`, `StillClock`, and `ImageViewer` (§§15.5, 15.9, 15.10, 15.12).

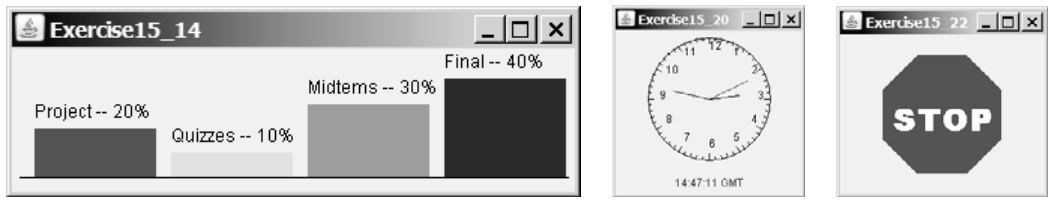


15.1 Introduction

Problem

Suppose you wish to draw shapes such as a bar chart, a clock, or a stop sign, as shown in Figure 15.1. How do you do so?

This chapter describes how to use the methods in the `Graphics` class to draw strings, lines, rectangles, ovals, arcs, polygons, and images, and how to develop reusable GUI components.



(a) (b) (c)

FIGURE 15.1 You can draw shapes using the drawing methods in the `Graphics` class.

15.2 Graphical Coordinate Systems

To paint, you need to specify where to paint. Each component has its own coordinate system with the origin (`0`, `0`) at the upper-left corner. The *x*-coordinate increases to the right, and the *y*-coordinate increases downward. Note that the Java coordinate system differs from the conventional coordinate system, as shown in Figure 15.2.

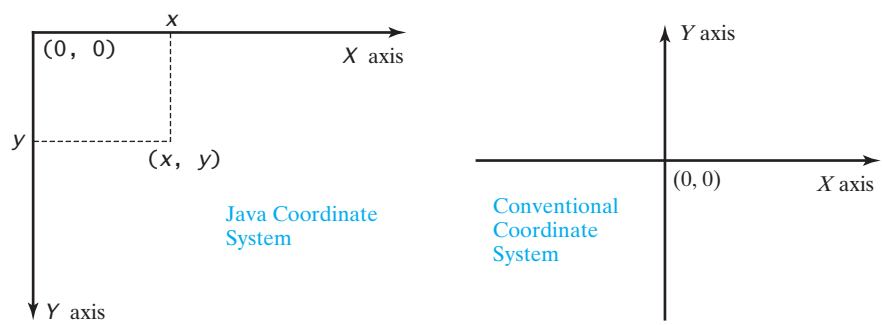


FIGURE 15.2 The Java coordinate system is measured in pixels, with `(0, 0)` at its upper-left corner.

The location of the upper-left corner of a component `c1` (e.g., a button) inside its parent component `c2` (e.g., a panel) can be located using `c1.getX()` and `c1.getY()`. As shown in Figure 15.3, `(x1, y1) = (c1.getX(), c1.getY())`, `(x2, y2) = (c2.getX(), c2.getY())`, and `(x3, y3) = (c3.getX(), c3.getY())`.

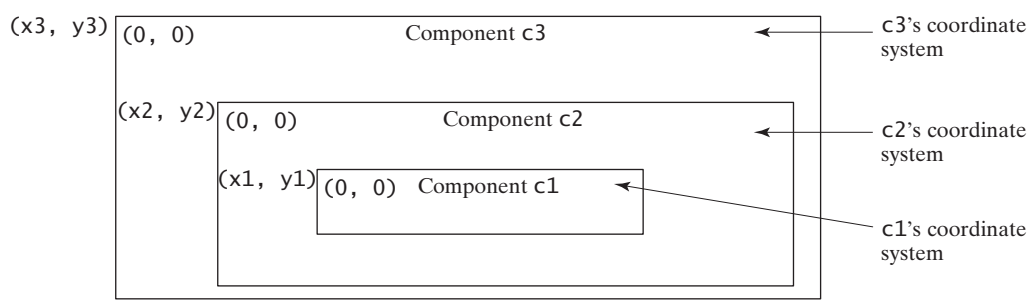


FIGURE 15.3 Each GUI component has its own coordinate system.

15.3 The Graphics Class

The `Graphics` class provides the methods for drawing strings, lines, rectangles, ovals, arcs, polygons, and polylines, as shown in Figure 15.4.

Think of a GUI component as a piece of paper and the `Graphics` object as a pencil or paintbrush. You can apply the methods in the `Graphics` class to draw things on a GUI component.

java.awt.Graphics	
`+setColor(color: Color): void`	Sets a new color for subsequent drawings.
`+setFont(font: Font): void`	Sets a new font for subsequent drawings.
`+drawString(s: String, x: int, y: int): void`	Draws a string starting at point `(x, y)`.
`+drawLine(x1: int, y1: int, x2: int, y2: int): void`	Draws a line from `(x1, y1)` to `(x2, y2)`.
`+drawRect(x: int, y: int, w: int, h: int): void`	Draws a rectangle with specified upper-left corner point at `(x,y)` and width `w` and height `h`.
`+fillRect(x: int, y: int, w: int, h: int): void`	Draws a filled rectangle with specified upper-left corner point at `(x, y)` and width `w` and height `h`.
`+drawRoundRect(x: int, y: int, w: int, h: int, aw: int, ah: int): void`	Draws a round-cornered rectangle with specified arc width `aw` and arc height `ah`.
`+fillRoundRect(x: int, y: int, w: int, h: int, aw: int, ah: int): void`	Draws a filled round-cornered rectangle with specified arc width `aw` and arc height `ah`.
`+draw3DRect(x: int, y: int, w: int, h: int, raised: boolean): void`	Draws a 3-D rectangle raised above the surface or sunk into the surface.
`+fill3DRect(x: int, y: int, w: int, h: int, raised: boolean): void`	Draws a filled 3-D rectangle raised above the surface or sunk into the surface.
`+drawOval(x: int, y: int, w: int, h: int): void`	Draws an oval bounded by the rectangle specified by the parameters `x`, `y`, `w`, and `h`.
`+fillOval(x: int, y: int, w: int, h: int): void`	Draws a filled oval bounded by the rectangle specified by the parameters `x`, `y`, `w`, and `h`.
`+drawArc(x: int, y: int, w: int, h: int, startAngle: int, arcAngle: int): void`	Draws an arc conceived as part of an oval bounded by the rectangle specified by the parameters `x`, `y`, `w`, and `h`.
`+fillArc(x: int, y: int, w: int, h: int, startAngle: int, arcAngle: int): void`	Draws a filled arc conceived as part of an oval bounded by the rectangle specified by the parameters `x`, `y`, `w`, and `h`.
`+drawPolygon(xPoints: int[], yPoints: int[], nPoints: int): void`	Draws a closed polygon defined by arrays of `x`- and `y`-coordinates. Each pair of `(x[i], y[i])`-coordinates is a point.
`+fillPolygon(xPoints: int[], yPoints: int[], nPoints: int): void`	Draws a filled polygon defined by arrays of `x`- and `y`-coordinates. Each pair of `(x[i], y[i])`-coordinates is a point.
`+drawPolygon(g: Polygon): void`	Draws a closed polygon defined by a `Polygon` object.
`+fillPolygon(g: Polygon): void`	Draws a filled polygon defined by a `Polygon` object.
`+drawPolyline(xPoints: int[], yPoints: int[], nPoints: int): void`	Draws a polyline defined by arrays of `x`- and `y`-coordinates. Each pair of `(x[i], y[i])`-coordinates is a point.

FIGURE 15.4 The `Graphics` class contains the methods for drawing strings and shapes.

The `Graphics` class–an abstract class—provides a device-independent graphics interface for displaying figures and images on the screen on different platforms. Whenever a component (e.g., a button, a label, a panel) is displayed, the JVM automatically creates a `Graphics` object for the component on the native platform and passes this object to invoke the `paintComponent` method to display the drawings.

The signature of the `paintComponent` method is as follows:

```
protected void paintComponent(Graphics g)
```

This method, defined in the `JComponent` class, is invoked whenever a component is first displayed or redisplayed.

In order to draw things on a component, you need to define a class that extends `JPanel` and overrides its `paintComponent` method to specify what to draw. Listing 15.1 gives an example that draws a line and a string on a panel, as shown in Figure 15.5.

LISTING 15.1 TestPaintComponent.java

```
1  import javax.swing.*;
2  import java.awt.Graphics;
3
4  public class TestPaintComponent extends JFrame {
5    public TestPaintComponent() {
6      add(new NewPanel());
7    }
8
9    public static void main(String[] args) {
10     TestPaintComponent frame = new TestPaintComponent();
11     frame.setTitle("TestPaintComponent");
12     frame.setSize(200, 100);
13     frame.setLocationRelativeTo(null); // Center the frame
14     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15     frame.setVisible(true);
16   }
17 }
18
19 class NewPanel extends JPanel {
20   protected void paintComponent(Graphics g) {
21     super.paintComponent(g);
22     g.drawLine(0, 0, 50, 50);
23     g.drawString("Banner", 0, 40);
24   }
25 }
```

create a panel

new panel class
override **paintComponent**
draw things in the superclass
draw line
draw string



FIGURE 15.5 A line and a string are drawn on a panel.

The `paintComponent` method is automatically invoked to paint graphics when the component is first displayed or whenever the component needs to be redisplayed. Invoking `super.paintComponent(g)` (line 21) invokes the `paintComponent` method defined in the superclass. This is necessary to ensure that the viewing area is cleared before a new drawing is displayed. Line 22 invokes the `drawLine` method to draw a line from (`0`, `0`) to (`50`, `50`). Line 23 invokes the `drawString` method to draw a string.

All the drawing methods have parameters that specify the locations of the subjects to be drawn. All measurements in Java are made in pixels. The string "Banner" is drawn at location (`0`, `40`).

The JVM invokes `paintComponent` to draw things on a component. The user should never invoke `paintComponent` directly. For this reason, the protected visibility is sufficient for `paintComponent`.

Panels are invisible and are used as small containers that group components to achieve a desired layout. Another important use of `JPanel` is for drawing. You can draw things on any Swing GUI component, but normally you should use a `JPanel` as a canvas upon which to draw things. What happens if you replace `JPanel` with `JLabel` in line 19 as follows?

```
class NewPanel extends JLabel {
```

The program will work, but it is not preferred, because `JLabel` is designed for creating a label, not for drawing. For consistency, this book will define a canvas class by subclassing `JPanel`.

`extends JPanel`?



Tip

Some textbooks define a canvas class by subclassing `JComponent`. The problem is that, if you wish to set a background in the canvas, you have to write the code to paint the background color. A simple `setBackground(Color color)` method will not set a background color in a `JComponent`.

`extends JComponent`?

15.4 Drawing Strings, Lines, Rectangles, and Ovals

The `drawString(String s, int x, int y)` method draws a string starting at the point `(x, y)`, as shown in Figure 15.6(a).

`drawString`

The `drawLine(int x1, int y1, int x2, int y2)` method draws a straight line from point `(x1, y1)` to point `(x2, y2)`, as shown in Figure 15.6(b).

`drawLine`

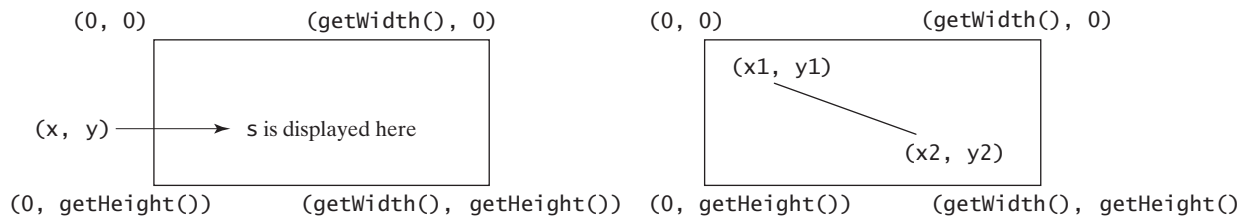


FIGURE 15.6 (a) The `drawString(s, x, y)` method draws a string starting at `(x, y)`. (b) The `drawLine(x1, y1, x2, y2)` method draws a line between two specified points.

Java provides six methods for drawing rectangles in outline or filled with color. You can draw or fill plain rectangles, round-cornered rectangles, or three-dimensional rectangles.

The `drawRect(int x, int y, int w, int h)` method draws a plain rectangle, and the `fillRect(int x, int y, int w, int h)` method draws a filled rectangle. The parameters `x` and `y` represent the upper-left corner of the rectangle, and `w` and `h` are its width and height (see Figure 15.7).

`drawRect`
`fillRect`

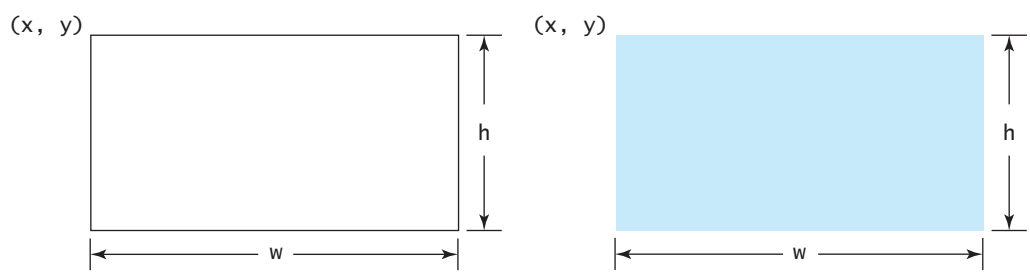


FIGURE 15.7 (a) The `drawRect(x, y, w, h)` method draws a rectangle. (b) The `fillRect(x, y, w, h)` method draws a filled rectangle.

The `drawRoundRect(int x, int y, int w, int h, int aw, int ah)` method draws a round-cornered rectangle, and the `fillRoundRect(int x, int y, int w, int h, int aw, int ah)` method draws a filled round-cornered rectangle. Parameters `x`, `y`, `w`, and `h` are the same as in the `drawRect` method, parameter `aw` is the horizontal diameter of the arcs at the

`drawRoundRect`
`fillRoundRect`

corner, and ah is the vertical diameter of the arcs at the corner (see Figure 15.8(a)). In other words, aw and ah are the width and the height of the oval that produces a quarter-circle at each corner.

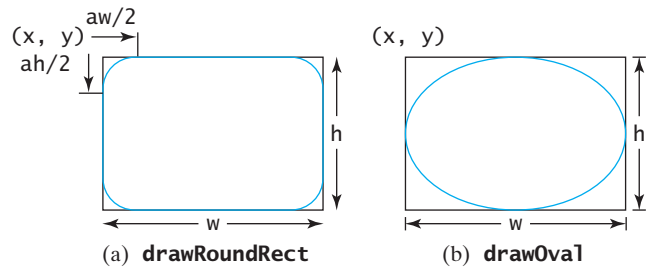


FIGURE 15.8 (a) The `drawRoundRect(x, y, w, h, aw, ah)` method draws a round-cornered rectangle. (b) The `drawOval(x, y, w, h)` method draws an oval based on its bounding rectangle.

`draw3DRect`
`fill3DRect`

The `draw3DRect(int x, int y, int w, int h, boolean raised)` method draws a 3D rectangle and the `fill3DRect(int x, int y, int w, int h, boolean raised)` method draws a filled 3D rectangle. The parameters `x`, `y`, `w`, and `h` are the same as in the `drawRect` method. The last parameter, a Boolean value, indicates whether the rectangle is raised above the surface or sunk into the surface.

`drawOval`
`fillOval`

Depending on whether you wish to draw an oval in outline or filled solid, you can use either the `drawOval(int x, int y, int w, int h)` method or the `fillOval(int x, int y, int w, int h)` method. An oval is drawn based on its bounding rectangle. Parameters `x` and `y` indicate the top-left corner of the bounding rectangle, and `w` and `h` indicate the width and height, respectively, of the bounding rectangle, as shown in Figure 15.8(b).

15.5 Case Study: The `FigurePanel` Class



Video Note
The `FigurePanel` class

This example develops a useful class for displaying various figures. The class enables the user to set the figure type and specify whether the figure is filled, and it displays the figure on a panel. The UML diagram for the class is shown in Figure 15.9. The panel can display lines, rectangles, round-cornered rectangles, and ovals. Which figure to display is decided by the `type` property. If the `filled` property is `true`, the rectangle, round-cornered rectangle, and oval are filled in the panel.

The UML diagram serves as the contract for the `FigurePanel` class. The user can use the class without knowing how the class is implemented. Let us begin by writing a program in Listing 15.2 that uses the class to display six figure panels, as shown in Figure 15.10.

LISTING 15.2 TestFigurePanel.java

create figures

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class TestFigurePanel extends JFrame {
5    public TestFigurePanel() {
6      setLayout(new GridLayout(2, 3, 5, 5));
7      add(new FigurePanel(FigurePanel.LINE));
8      add(new FigurePanel(FigurePanel.RECTANGLE));
9      add(new FigurePanel(FigurePanel.ROUND_RECTANGLE));
10     add(new FigurePanel(FigurePanel.OVAL));
11     add(new FigurePanel(FigurePanel.RECTANGLE, true));
```

```
12      add(new FigurePanel(FigurePanel.ROUND_RECTANGLE, true));
13    }
14
15    public static void main(String[] args) {
16      TestFigurePanel frame = new TestFigurePanel();
17      frame.setSize(400, 200);
18      frame.setTitle("TestFigurePanel");
19      frame.setLocationRelativeTo(null); // Center the frame
20      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21      frame.setVisible(true);
22    }
23  }
```

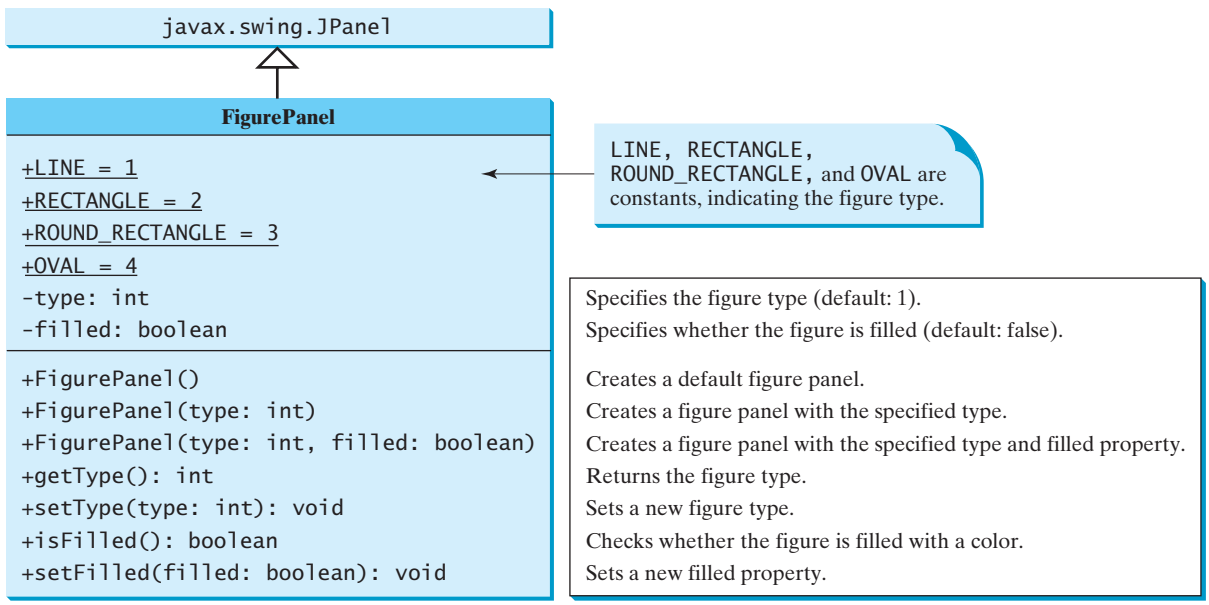

javax.swing.JPanel
FigurePanel
+LINE = 1
+RECTANGLE = 2
+ROUND_RECTANGLE = 3
+OVAL = 4
-type: int
-filled: boolean
+FigurePanel()
+FigurePanel(type: int)
+FigurePanel(type: int, filled: boolean)
+getType(): int
+setType(type: int): void
+isFilled(): boolean
+setFilled(filled: boolean): void
LINE, RECTANGLE, ROUND_RECTANGLE, and OVAL are constants, indicating the figure type.
Specifies the figure type (default: 1).
Specifies whether the figure is filled (default: false).
Creates a default figure panel.
Creates a figure panel with the specified type.
Creates a figure panel with the specified type and filled property.
Returns the figure type.
Sets a new figure type.
Checks whether the figure is filled with a color.
Sets a new filled property.

FIGURE 15.9 `FigurePanel` displays various types of figures on the panel.

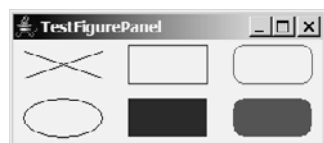


FIGURE 15.10 Six `FigurePanel` objects are created to display six figures.

The `FigurePanel` class is implemented in Listing 15.3. Four constants—`LINE`, `RECTANGLE`, `ROUND_RECTANGLE`, and `OVAL`—are declared in lines 6–9. Four types of figures are drawn according to the `type` property (line 37). The `setColor` method (lines 39, 44, 53, 62) sets a new color for the drawing.

LISTING 15.3 FigurePanel.java

```
1 import java.awt.*;
2 import javax.swing.JPanel;
3
4 public class FigurePanel extends JPanel {
```

constants

override **paintComponent(g)**

check type

draw lines

fill a rectangle

draw a rectangle

fill round-cornered rect

draw round-cornered rect

fill an oval

```
5   // Define constants
6   public static final int LINE = 1;
7   public static final int RECTANGLE = 2;
8   public static final int ROUND_RECTANGLE = 3;
9   public static final int OVAL = 4;
10
11  private int type = 1;
12  private boolean filled = false;
13
14  /** Construct a default FigurePanel */
15  public FigurePanel() {
16  }
17
18  /** Construct a FigurePanel with the specified type */
19  public FigurePanel(int type) {
20    this.type = type;
21  }
22
23  /** Construct a FigurePanel with the specified type and filled */
24  public FigurePanel(int type, boolean filled) {
25    this.type = type;
26    this.filled = filled;
27  }
28
29  /** Draw a figure on the panel */
30  protected void paintComponent(Graphics g) {
31    super.paintComponent(g);
32
33    // Get the appropriate size for the figure
34    int width = getWidth();
35    int height = getHeight();
36
37    switch (type) {
38      case LINE: // Display two cross lines
39        g.setColor(Color.BLACK);
40        g.drawLine(10, 10, width - 10, height - 10);
41        g.drawLine(width - 10, 10, 10, height - 10);
42        break;
43      case RECTANGLE: // Display a rectangle
44        g.setColor(Color.BLUE);
45        if (filled)
46          g.fillRect((int)(0.1 * width), (int)(0.1 * height),
47            (int)(0.8 * width), (int)(0.8 * height));
48        else
49          g.drawRect((int)(0.1 * width), (int)(0.1 * height),
50            (int)(0.8 * width), (int)(0.8 * height));
51        break;
52      case ROUND_RECTANGLE: // Display a round-cornered rectangle
53        g.setColor(Color.RED);
54        if (filled)
55          g.fillRoundRect((int)(0.1 * width), (int)(0.1 * height),
56            (int)(0.8 * width), (int)(0.8 * height), 20, 20);
57        else
58          g.drawRoundRect((int)(0.1 * width), (int)(0.1 * height),
59            (int)(0.8 * width), (int)(0.8 * height), 20, 20);
60        break;
61      case OVAL: // Display an oval
62        g.setColor(Color.BLACK);
63        if (filled)
64          g.fillOval((int)(0.1 * width), (int)(0.1 * height),
```

```
65            (int)(0.8 * width), (int)(0.8 * height));
66        else
67          g.drawOval((int)(0.1 * width), (int)(0.1 * height),
68            (int)(0.8 * width), (int)(0.8 * height));
69    }
70  }
71
72  /** Set a new figure type */
73  public void setType(int type) {
74    this.type = type;
75    repaint();
76  }
77
78  /** Return figure type */
79  public int getType() {
80    return type;
81  }
82
83  /** Set a new filled property */
84  public void setFilled(boolean filled) {
85    this.filled = filled;
86    repaint();
87  }
88
89  /** Check if the figure is filled */
90  public boolean isFilled() {
91    return filled;
92  }
93
94  /** Specify preferred size */
95  public Dimension getPreferredSize() {
96    return new Dimension(80, 80);
97  }
98 }
```

draw an oval

repaint panel

repaint panel

override **getPreferredSize()**

The **repaint** method (lines 75, 86) is defined in the **Component** class. Invoking **repaint** causes the **paintComponent** method to be called. The **repaint** method is invoked to refresh the viewing area. Typically, you call it if you have new things to display.



Caution

The **paintComponent** method should never be invoked directly. It is invoked either by the JVM whenever the viewing area changes or by the **repaint** method. You should override the **paintComponent** method to tell the system how to paint the viewing area, but never override the **repaint** method.

don't invoke **paintComponent**



Note

The **repaint** method lodges a request to update the viewing area and returns immediately. Its effect is asynchronous, meaning that it is up to the JVM to execute the **paintComponent** method on a separate thread.

request repaint using **repaint()**

The **getPreferredSize()** method (lines 95–97), defined in **Component**, is overridden in **FigurePanel** to specify the preferred size for the layout manager to consider when laying out a **FigurePanel** object. This property may or may not be considered by the layout manager, depending on its rules. For example, a component uses its preferred size in a container with a **FlowLayout** manager, but its preferred size may be ignored if it is placed in a container with a **GridLayout** manager. It is a good practice to override **getPreferredSize()** in a subclass of **JPanel** to specify a preferred size, because the default preferred size for a **JPanel** is **0** by **0**.

getPreferedSize()

15.6 Drawing Arcs

An arc is conceived as part of an oval bounded by a rectangle. The methods to draw or fill an arc are as follows:

```
drawArc(int x, int y, int w, int h, int startAngle, int arcAngle);
fillArc(int x, int y, int w, int h, int startAngle, int arcAngle);
```

Parameters `x`, `y`, `w`, and `h` are the same as in the `drawOval` method; parameter `startAngle` is the starting angle; `arcAngle` is the spanning angle (i.e., the angle covered by the arc). Angles are measured in degrees and follow the usual mathematical conventions (i.e., `0` degrees is in the easterly direction, and positive angles indicate counterclockwise rotation from the easterly direction); see Figure 15.11.

Listing 15.4 is an example of how to draw arcs; the output is shown in Figure 15.12.

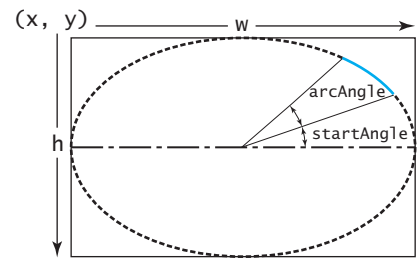


FIGURE 15.11 The `drawArc` method draws an arc based on an oval with specified angles.

LISTING 15.4 DrawArcs.java

```
1 import javax.swing.JFrame;
2 import javax.swing.JPanel;
3 import java.awt.Graphics;
4
5 public class DrawArcs extends JFrame {
6   public DrawArcs() {
7     setTitle("DrawArcs");
8     add(new ArcsPanel());
9   }
10
11  /** Main method */
12  public static void main(String[] args) {
13    DrawArcs frame = new DrawArcs();
14    frame.setSize(250, 300);
15    frame.setLocationRelativeTo(null); // Center the frame
16    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17    frame.setVisible(true);
18  }
19 }
20
21 // The class for drawing arcs on a panel
22 class ArcsPanel extends JPanel {
23   // Draw four blades of a fan
24   protected void paintComponent(Graphics g) {
25     super.paintComponent(g);
26
27     int xCenter = getWidth() / 2;
28     int yCenter = getHeight() / 2;
29     int radius = (int)(Math.min(getWidth(), getHeight()) * 0.4);
30
```

add a panel

override **paintComponent**

```
31      int x = xCenter - radius;
32      int y = yCenter - radius;
33
34      g.fillArc(x, y, 2 * radius, 2 * radius, 0, 30);
35      g.fillArc(x, y, 2 * radius, 2 * radius, 90, 30);
36      g.fillArc(x, y, 2 * radius, 2 * radius, 180, 30);
37      g.fillArc(x, y, 2 * radius, 2 * radius, 270, 30);
38    }
39 }
```

30° arc from 0°
30° arc from 90°
30° arc from 180°
30° arc from 270°

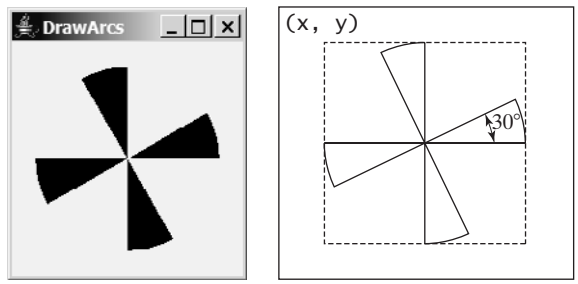


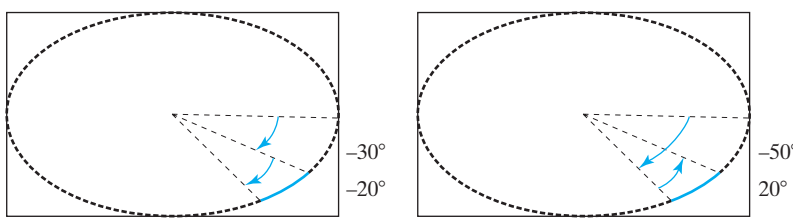
FIGURE 15.12 The program draws four filled arcs.

negative degrees

Angles may be negative. A negative starting angle sweeps clockwise from the easterly direction, as shown in Figure 15.13. A negative spanning angle sweeps clockwise from the starting angle. The following two statements draw the same arc:

```
g.fillArc(x, y, 2 * radius, 2 * radius, -30, -20);
g.fillArc(x, y, 2 * radius, 2 * radius, -50, 20);
```

The first statement uses negative starting angle `-30` and negative spanning angle `-20`, as shown in Figure 15.13(a). The second statement uses negative starting angle `-50` and positive spanning angle `20`, as shown in Figure 15.13(b).



(a) Negative starting angle –30° and negative spanning angle –20°

(b) Negative starting angle –50° and positive spanning angle 20°

FIGURE 15.13 Angles may be negative.

15.7 Drawing Polygons and Polylines

To draw a polygon, first create a `Polygon` object using the `Polygon` class, as shown in Figure 15.14.

A polygon is a closed two-dimensional region. This region is bounded by an arbitrary number of line segments, each being one side (or edge) of the polygon. A polygon comprises a list of `(x, y)`-coordinate pairs in which each pair defines a vertex of the polygon, and two successive pairs are the endpoints of a line that is a side of the polygon. The first and final points are joined by a line segment that closes the polygon.

Here is an example of creating a polygon and adding points into it:

```
Polygon polygon = new Polygon();
polygon.addPoint(40, 20);
```

```
polygon.addPoint(70, 40);
polygon.addPoint(60, 80);
polygon.addPoint(45, 45);
polygon.addPoint(20, 60);
```

java.awt.Polygon	
+xpoints: int[]	x-coordinates of all points in the polygon.
+ypoints: int[]	y-coordinates of all points in the polygon.
+npoints: int	The number of points in the polygon.
+Polygon()	Creates an empty polygon.
+Polygon(xpoints: int[], ypoints: int[], npoints: int)	Creates a polygon with the specified points.
+addPoint(x: int, y: int)	Appends a point to the polygon.

FIGURE 15.14 The `Polygon` class models a polygon.

After these points are added, `xpoints` is {40, 70, 60, 45, 20}, `ypoints` is {20, 40, 80, 45, 60}, and `npoints` is 5. `xpoints`, `ypoints`, and `npoints` are public data fields in `Polygon`, which is a bad design. In principle, all data fields should be kept private.

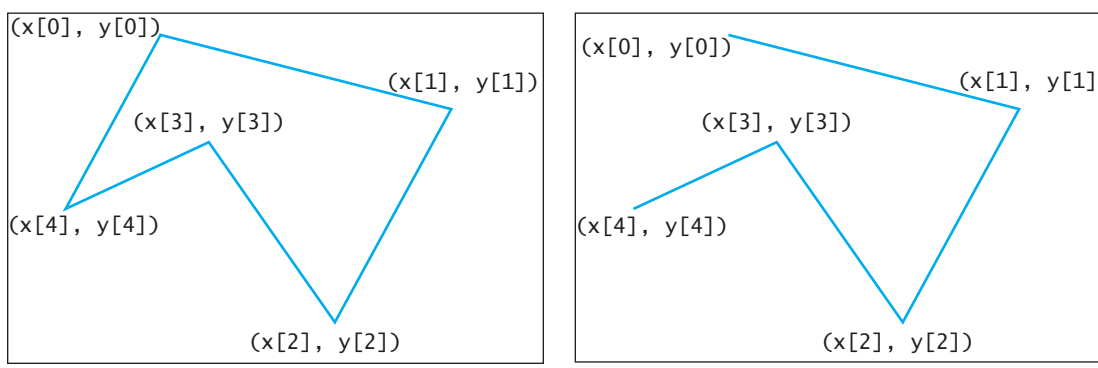
To draw or fill a polygon, use one of the following methods in the `Graphics` class:

```
drawPolygon(Polygon polygon);
fillPolygon(Polygon polygon);
drawPolygon(int[] xpoints, int[] ypoints, int npoints);
fillPolygon(int[] xpoints, int[] ypoints, int npoints);
```

For example:

```
int x[] = {40, 70, 60, 45, 20};
int y[] = {20, 40, 80, 45, 60};
g.drawPolygon(x, y, x.length);
```

The drawing method opens the polygon by drawing lines between point `(x[i], y[i])` and point `(x[i+1], y[i+1])` for `i = 0, ... , x.length-1`; it closes the polygon by drawing a line between the first and last points (see Figure 15.15(a)).



(a) Polygon (b) Polyline

FIGURE 15.15 The `drawPolygon` method draws a polygon, and the `polyLine` method draws a polyline.

To draw a polyline, use the **drawPolyline(int[] x, int[] y, int nPoints)** method, which draws a sequence of connected lines defined by arrays of **x-** and **y**-coordinates. For example, the following code draws the polyline, as shown in Figure 15.15(b).

```
int x[] = {40, 70, 60, 45, 20};
int y[] = {20, 40, 80, 45, 60};
g.drawPolyline(x, y, x.length);
```

Listing 15.5 is an example of how to draw a hexagon, with the output shown in Figure 15.16.

LISTING 15.5 DrawPolygon.java

```
 1 import javax.swing.JFrame;
 2 import javax.swing.JPanel;
 3 import java.awt.Graphics;
 4 import java.awt.Polygon;
 5
 6 public class DrawPolygon extends JFrame {
 7   public DrawPolygon() {
 8     setTitle("DrawPolygon");
 9     add(new PolygonsPanel());                                    add a panel
10   }
11
12   /** Main method */
13   public static void main(String[] args) {
14     DrawPolygon frame = new DrawPolygon();
15     frame.setSize(200, 250);
16     frame.setLocationRelativeTo(null); // Center the frame
17     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18     frame.setVisible(true);
19   }
20 }
21
22 // Draw a polygon in the panel
23 class PolygonsPanel extends JPanel {
24   protected void paintComponent(Graphics g) {                    paintComponent
25     super.paintComponent(g);
26
27     int xCenter = getWidth() / 2;
28     int yCenter = getHeight() / 2;
29     int radius = (int)(Math.min(getWidth(), getHeight()) * 0.4);
30
31     // Create a Polygon object
32     Polygon polygon = new Polygon();
33
34     // Add points to the polygon in this order
35     polygon.addPoint(xCenter + radius, yCenter);                 add a point
36     polygon.addPoint((int)(xCenter + radius *
37       Math.cos(2 * Math.PI / 6)), (int)(yCenter - radius *
38       Math.sin(2 * Math.PI / 6)));
39     polygon.addPoint((int)(xCenter + radius *
40       Math.cos(2 * 2 * Math.PI / 6)), (int)(yCenter - radius *
41       Math.sin(2 * 2 * Math.PI / 6)));
42     polygon.addPoint((int)(xCenter + radius *
43       Math.cos(3 * 2 * Math.PI / 6)), (int)(yCenter - radius *
44       Math.sin(3 * 2 * Math.PI / 6)));
45     polygon.addPoint((int)(xCenter + radius *
46       Math.cos(4 * 2 * Math.PI / 6)), (int)(yCenter - radius *
47       Math.sin(4 * 2 * Math.PI / 6)));
48     polygon.addPoint((int)(xCenter + radius *
```

```
49        Math.cos(5 * 2 * Math.PI / 6)), (int)(yCenter - radius *
50        Math.sin(5 * 2 * Math.PI / 6)));
51
52      // Draw the polygon
53      g.drawPolygon(polygon);
54    }
55  }
```

draw polygon

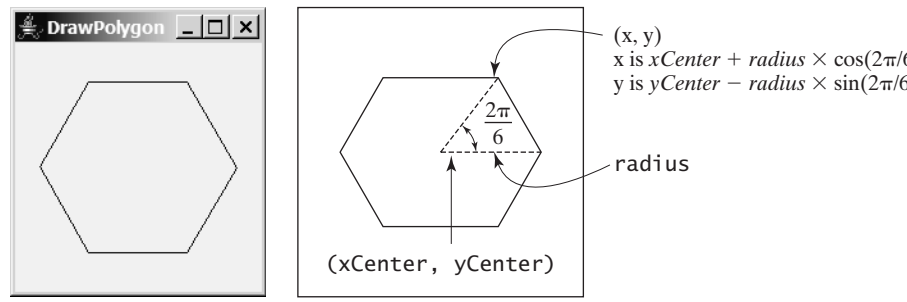


FIGURE 15.16 The program uses the `drawPolygon` method to draw a polygon.

15.8 Centering a String Using the `FontMetrics` Class

You can display a string at any location in a panel. Can you display it centered? To do so, you need to use the `FontMetrics` class to measure the exact width and height of the string for a particular font. `FontMetrics` can measure the following attributes for a given font (see Figure 15.17):

- **Leading,** pronounced *ledding*, is the amount of space between lines of text.
- **Ascent** is the distance from the baseline to the ascent line. The top of most characters in the font will be under the ascent line, but some may extend above the ascent line.
- **Descent** is the distance from the baseline to the descent line. The bottom of most descending characters (e.g., *j*, *y*, and *g*) in the font will be above the descent line, but some may extend below the descent line.
- **Height** is the sum of leading, ascent, and descent.

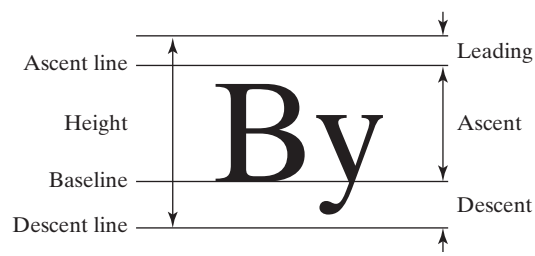


FIGURE 15.17 The `FontMetrics` class can be used to determine the font properties of characters for a given font.

`FontMetrics` is an abstract class. To get a `FontMetrics` object for a specific font, use the following `getFontMetrics` methods defined in the `Graphics` class:

- `public FontMetrics getFontMetrics(Font font)`
  Returns the font metrics of the specified font.
- `public FontMetrics getFontMetrics()`
  Returns the font metrics of the current font.

You can use the following instance methods in the **FontMetrics** class to obtain the attributes of a font and the width of a string when it is drawn using the font:

```
public int getAscent() // Return the ascent
public int getDescent() // Return the descent
public int getLeading() // Return the leading
public int getHeight() // Return the height
public int stringWidth(String str) // Return the width of the string
```

Listing 15.6 gives an example that displays a message in the center of the panel, as shown in Figure 15.18.

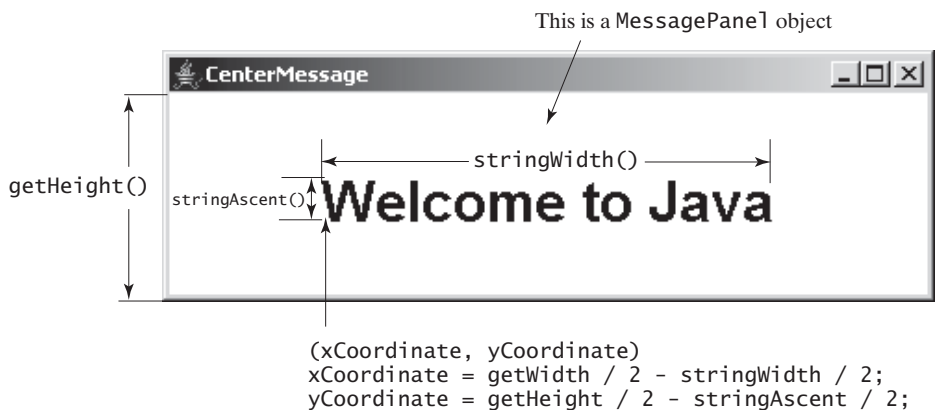


FIGURE 15.18 The program uses the **FontMetrics** class to measure the string width and height and displays it at the center of the panel.

LISTING 15.6 TestCenterMessage.java

```
1 import javax.swing.*;
2 import java.awt.*;
3
4 public class TestCenterMessage extends JFrame{
5   public TestCenterMessage() {
6     CenterMessage messagePanel = new CenterMessage();          create a message panel
7     add(messagePanel);                                         add a message panel
8     messagePanel.setBackground(Color.WHITE);                   set background
9     messagePanel.setFont(new Font("Californian FB", Font.BOLD, 30));   set font
10  }
11
12  /** Main method */
13  public static void main(String[] args) {
14    TestCenterMessage frame = new TestCenterMessage();
15    frame.setSize(300, 150);
16    frame.setLocationRelativeTo(null); // Center the frame
17    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18    frame.setVisible(true);
19  }
20 }
21
22 class CenterMessage extends JPanel {
23  /** Paint the message */
24  protected void paintComponent(Graphics g) {                  override paintComponent
25    super.paintComponent(g);
```

get FontMetrics

```
26
27      // Get font metrics for the current font
28      FontMetrics fm = g.getFontMetrics();
29
30      // Find the center location to display
31      int stringWidth = fm.stringWidth("Welcome to Java");
32      int stringAscent = fm.getAscent();
33
34      // Get the position of the leftmost character in the baseline
35      int xCoordinate = getWidth() / 2 - stringWidth / 2;
36      int yCoordinate = getHeight() / 2 + stringAscent / 2;
37
38      g.drawString("Welcome to Java", xCoordinate, yCoordinate);
39    }
40 }
```

The methods `getWidth()` and `getHeight()` (lines 35–36) defined in the `Component` class return the component's width and height, respectively.

Since the message is `centered`, the first character of the string should be positioned at (`xCoordinate`, `yCoordinate`), as shown in Figure 15.18.

15.9 Case Study: The `MessagePanel` Class



Video Note
The `MessagePanel` class

This case study develops a useful class that displays a message in a panel. The class enables the user to set the location of the message, center the message, and move the message with the specified interval. The contract of the class is shown in Figure 15.19.

Let us first write a test program in Listing 15.7 that uses the `MessagePanel` class to display four message panels, as shown in Figure 15.20.

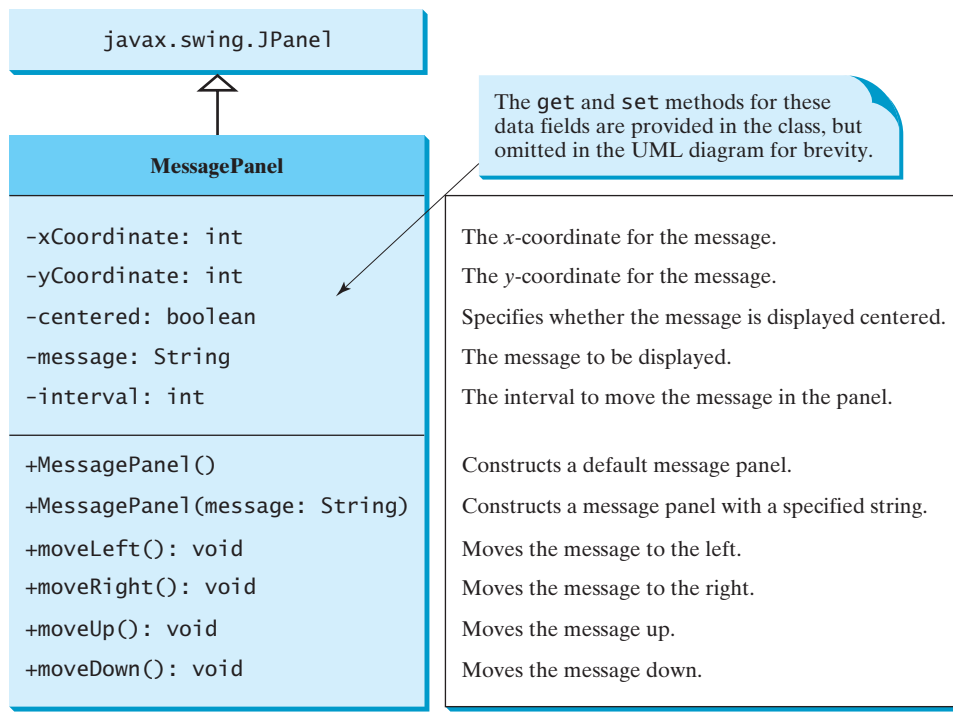

javax.swing.JPanel
MessagePanel
-xCoordinate: int
-yCoordinate: int
-centered: boolean
-message: String
-interval: int
+MessagePanel()
+MessagePanel(message: String)
+moveLeft(): void
+moveRight(): void
+moveUp(): void
+moveDown(): void
The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
The x-coordinate for the message.
The y-coordinate for the message.
Specifies whether the message is displayed centered.
The message to be displayed.
The interval to move the message in the panel.
Constructs a default message panel.
Constructs a message panel with a specified string.
Moves the message to the left.
Moves the message to the right.
Moves the message up.
Moves the message down.

FIGURE 15.19 `MessagePanel` displays a message on the panel.

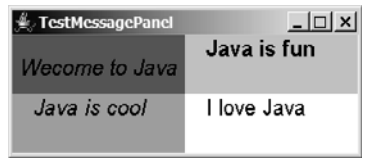


FIGURE 15.20 `TestMessagePanel` uses `MessagePanel` to display four message panels.

LISTING 15.7 TestMessagePanel.java

```
 1 import java.awt.*;
 2 import javax.swing.*;
 3
 4 public class TestMessagePanel extends JFrame {
 5   public TestMessagePanel() {
 6     MessagePanel messagePanel1 = new MessagePanel("Wecome to Java");
 7     MessagePanel messagePanel2 = new MessagePanel("Java is fun");
 8     MessagePanel messagePanel3 = new MessagePanel("Java is cool");
 9     MessagePanel messagePanel4 = new MessagePanel("I love Java");
10     messagePanel1.setFont(new Font("SansSerif", Font.ITALIC, 20));
11     messagePanel2.setFont(new Font("Courier", Font.BOLD, 20));
12     messagePanel3.setFont(new Font("Times", Font.ITALIC, 20));
13     messagePanel4.setFont(new Font("Californian FB", Font.PLAIN, 20));
14     messagePanel1.setBackground(Color.RED);
15     messagePanel2.setBackground(Color.CYAN);
16     messagePanel3.setBackground(Color.GREEN);
17     messagePanel4.setBackground(Color.WHITE);
18     messagePanel1.setCentered(true);
19
20     setLayout(new GridLayout(2, 2));
21     add(messagePanel1);
22     add(messagePanel2);
23     add(messagePanel3);
24     add(messagePanel4);
25   }
26
27   public static void main(String[] args) {
28     TestMessagePanel frame = new TestMessagePanel();
29     frame.setSize(300, 200);
30     frame.setTitle("TestMessagePanel");
31     frame.setLocationRelativeTo(null); // Center the frame
32     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
33     frame.setVisible(true);
34   }
35 }
```

create message panel

set font

set background

add message panel

The rest of this section explains how to implement the `MessagePanel` class. Since you can use the class without knowing how it is implemented, you may skip the implementation if you wish.

skip implementation?

The `MessagePanel` class is implemented in Listing 15.8. The program seems long but is actually simple, because most of the methods are `get` and `set` methods, and each method is relatively short and easy to read.

LISTING 15.8 MessagePanel.java

```
1 import java.awt.FontMetrics;
2 import java.awt.Dimension;
3 import java.awt.Graphics;
```

```
 4 import javax.swing.JPanel;
 5
 6 public class MessagePanel extends JPanel {
 7   /** The message to be displayed */
 8   private String message = "Welcome to Java";
 9
10   /** The x-coordinate where the message is displayed */
11   private int xCoordinate = 20;
12
13   /** The y-coordinate where the message is displayed */
14   private int yCoordinate = 20;
15
16   /** Indicate whether the message is displayed in the center */
17   private boolean centered;
18
19   /** The interval for moving the message horizontally and
20     vertically */
21   private int interval = 10;
22
23   /** Construct with default properties */
24   public MessagePanel() {
25   }
26
27   /** Construct a message panel with a specified message */
28   public MessagePanel(String message) {
29     this.message = message;
30   }
31
32   /** Return message */
33   public String getMessage() {
34     return message;
35   }
36
37   /** Set a new message */
38   public void setMessage(String message) {
39     this.message = message;
40     repaint();
41   }
42
43   /** Return xCoordinator */
44   public int getXCoordinate() {
45     return xCoordinate;
46   }
47
48   /** Set a new xCoordinator */
49   public void setXCoordinate(int x) {
50     this.xCoordinate = x;
51     repaint();
52   }
53
54   /** Return yCoordinator */
55   public int getYCoordinate() {
56     return yCoordinate;
57   }
58
59   /** Set a new yCoordinator */
60   public void setYCoordinate(int y) {
61     this.yCoordinate = y;
```

repaint panel

repaint panel

```
62      repaint();                                                    repaint panel
63    }
64
65    /** Return centered */
66    public boolean isCentered() {
67      return centered;
68    }
69
70    /** Set a new centered */
71    public void setCentered(boolean centered) {
72      this.centered = centered;
73      repaint();                                                    repaint panel
74    }
75
76    /** Return interval */
77    public int getInterval() {
78      return interval;
79    }
80
81    /** Set a new interval */
82    public void setInterval(int interval) {
83      this.interval = interval;
84      repaint();                                                    repaint panel
85    }
86
87    /** Paint the message */
88    protected void paintComponent(Graphics g) {                     override paintComponent
89      super.paintComponent(g);
90
91      if (centered) {                                               check centered
92        // Get font metrics for the current font
93        FontMetrics fm = g.getFontMetrics();
94
95        // Find the center location to display
96        int stringWidth = fm.stringWidth(message);
97        int stringAscent = fm.getAscent();
98        // Get the position of the leftmost character in the baseline
99        xCoordinate = getWidth() / 2 - stringWidth / 2;
100       yCoordinate = getHeight() / 2 + stringAscent / 2;
101     }
102
103     g.drawString(message, xCoordinate, yCoordinate);
104   }
105
106   /** Move the message left */
107   public void moveLeft() {
108     xCoordinate -= interval;
109     repaint();
110   }
111
112   /** Move the message right */
113   public void moveRight() {
114     xCoordinate += interval;
115     repaint();
116   }
117
118   /** Move the message up */
119   public void moveUp() {
120     yCoordinate -= interval;
```

```
121     repaint();
122   }
123
124   /** Move the message down */
125   public void moveDown() {
126     yCoordinate += interval;
127     repaint();
128   }
129
130   /** Override get method for preferredSize */
131   public Dimension getPreferredSize() {
132     return new Dimension(200, 30);
133   }
134 }
```

override `getPreferredSize`

The `paintComponent` method displays the message centered, if the `centered` property is `true` (line 91). `message` is initialized to "Welcome to Java" in line 8. If it were not initialized, a `NullPointerException` runtime error would occur when you created a `MessagePanel` using the no-arg constructor, because `message` would be `null` in line 103.



Caution

The `MessagePanel` class uses the properties `xCoordinate` and `yCoordinate` to specify the position of the message displayed on the panel. Do not use the property names `x` and `y`, because they are already defined in the `Component` class to return the position of the component in the parent's coordinate system using `getX()` and `getY()`.



Note

The `Component` class has the `setBackground`, `setForeground`, and `setFont` methods. These methods are for setting colors and fonts for the entire component. Suppose you want to draw several messages in a panel with different colors and fonts; you have to use the `setColor` and `setFont` methods in the `Graphics` class to set the color and font for the current drawing.



Note

design classes for reuse

A key feature of Java programming is the reuse of classes. Throughout this book, reusable classes are developed and later reused. `MessagePanel` is an example, as are `Loan` in Listing 10.2 and `FigurePanel` in Listing 15.3. `MessagePanel` can be reused whenever you need to display a message on a panel. To make your class reusable in a wide range of applications, you should provide a variety of ways to use it. `MessagePanel` provides many properties and methods that will be used in several examples in the book. The next section presents a useful and reusable class for displaying a clock on a panel graphically.

15.10 Case Study: The `StillClock` Class



Video Note
The `StillClock` class

This case study develops a class that displays a clock on a panel. The contract of the class is shown in Figure 15.21.

Let us first write a test program in Listing 15.9 that uses the `StillClock` class to display an analog clock and uses the `MessagePanel` class to display the hour, minute, and second in a panel, as shown in Figure 15.22(a).

LISTING 15.9 DisplayClock.java

```
1 import java.awt.*;
2 import javax.swing.*;
3
4 public class DisplayClock extends JFrame {
5   public DisplayClock() {
```

```
6     // Create an analog clock for the current time
7     StillClock clock = new StillClock();                          create a clock
8
9     // Display hour, minute, and second in the message panel
10    MessagePanel messagePanel = new MessagePanel(clock.getHour() +  create a message panel
11      ":" + clock.getMinute() + ":" + clock.getSecond());
12    messagePanel.setCentered(true);
13    messagePanel.setForeground(Color.blue);
14    messagePanel.setFont(new Font("Courier", Font.BOLD, 16));
15
16    // Add the clock and message panel to the frame
17    add(clock);                                                   add a clock
18    add(messagePanel, BorderLayout.SOUTH);                        add a message panel
19  }
20
21  public static void main(String[] args) {
22    DisplayClock frame = new DisplayClock();
23    frame.setTitle("DisplayClock");
24    frame.setSize(300, 350);
25    frame.setLocationRelativeTo(null); // Center the frame
26    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27    frame.setVisible(true);
28  }
29 }
```

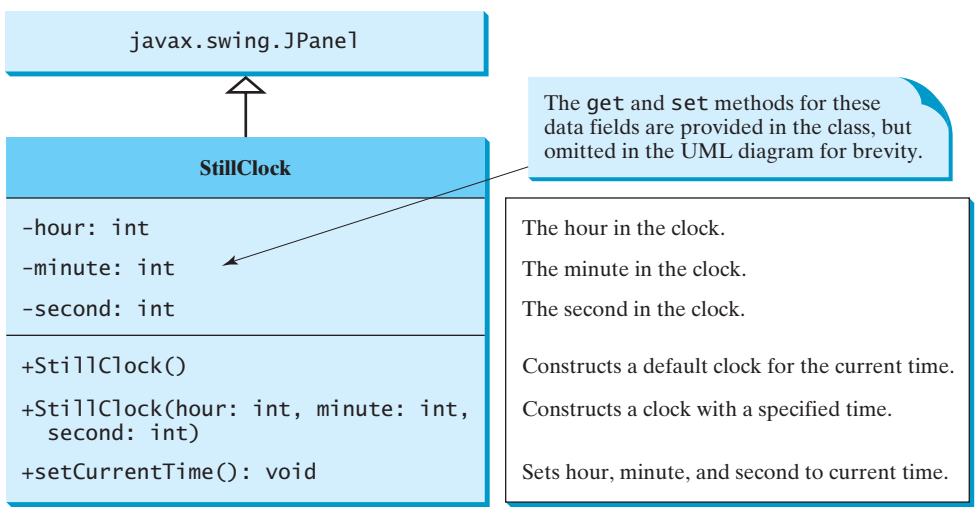

javax.swing.JPanel
StillClock
-hour: int
-minute: int
-second: int
+StillClock()
+StillClock(hour: int, minute: int, second: int)
+setCurrentTime(): void
The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
The hour in the clock.
The minute in the clock.
The second in the clock.
Constructs a default clock for the current time.
Constructs a clock with a specified time.
Sets hour, minute, and second to current time.

FIGURE 15.21 `StillClock` displays an analog clock.

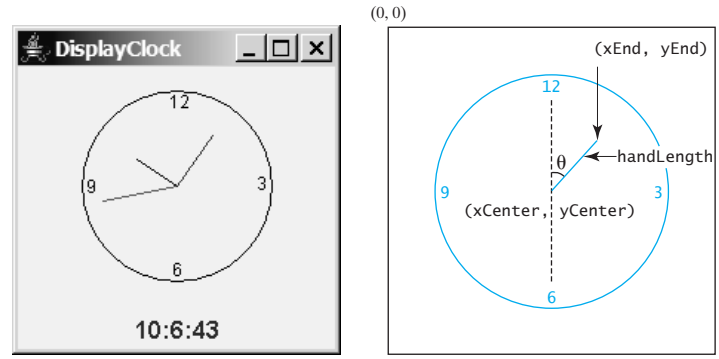
The rest of this section explains how to implement the `StillClock` class. Since you can use the class without knowing how it is implemented, you may skip the implementation if you wish. skip implementation?

To draw a clock, you need to draw a circle and three hands for second, minute, and hour. implementation To draw a hand, you need to specify the two ends of the line. As shown in Figure 15.22(b), one end is the center of the clock at `(xCenter, yCenter)`; the other end, at `(xEnd, yEnd)`, is determined by the following formula:

$$xEnd = xCenter + handLength \times \sin(\theta)$$
$$yEnd = yCenter - handLength \times \cos(\theta)$$

Since there are 60 seconds in one minute, the angle for the second hand is

$$second \times (2\pi/60)$$



(a) (b)

FIGURE 15.22 (a) The DisplayClock program displays a clock that shows the current time. (b) The endpoint of a clock hand can be determined, given the spanning angle, the hand length, and the center point.

The position of the minute hand is determined by the minute and second. The exact minute value combined with seconds is `minute + second/60`. For example, if the time is 3 minutes and 30 seconds, the total minutes are 3.5. Since there are 60 minutes in one hour, the angle for the minute hand is

$$(\text{minute} + \text{second}/60) \times (2\pi/60)$$

Since one circle is divided into 12 hours, the angle for the hour hand is

$$(\text{hour} + \text{minute}/60 + \text{second}/(60 \times 60)) \times (2\pi/12)$$

For simplicity in computing the angles of the minute hand and hour hand, you can omit the seconds, because they are negligibly small. Therefore, the endpoints for the second hand, minute hand, and hour hand can be computed as:

$$\begin{aligned}
\text{xSecond} &= \text{xCenter} + \text{secondHandLength} \times \sin(\text{second} \times (2\pi/60)) \\
\text{ySecond} &= \text{yCenter} - \text{secondHandLength} \times \cos(\text{second} \times (2\pi/60)) \\
\text{xMinute} &= \text{xCenter} + \text{minuteHandLength} \times \sin(\text{minute} \times (2\pi/60)) \\
\text{yMinute} &= \text{yCenter} - \text{minuteHandLength} \times \cos(\text{minute} \times (2\pi/60)) \\
\text{xHour} &= \text{xCenter} + \text{hourHandLength} \times \sin((\text{hour} + \text{minute}/60) \times (2\pi/60)) \\
\text{yHour} &= \text{yCenter} - \text{hourHandLength} \times \cos((\text{hour} + \text{minute}/60) \times (2\pi/60))
\end{aligned}$$

The `StillClock` class is implemented in Listing 15.10.

LISTING 15.10 StillClock.java

```
1  import java.awt.*;
2  import javax.swing.*;
3  import java.util.*;
4
5  public class StillClock extends JPanel {
6    private int hour;
7    private int minute;
8    private int second;
9
10   /** Construct a default clock with the current time*/
11   public StillClock() {
12     setCurrentTime();
13   }
14
15   /** Construct a clock with specified hour, minute, and second */
```

```
16  public StillClock(int hour, int minute, int second) {
17    this.hour = hour;
18    this.minute = minute;
19    this.second = second;
20  }
21
22  /** Return hour */
23  public int getHour() {
24    return hour;
25  }
26
27  /** Set a new hour */
28  public void setHour(int hour) {
29    this.hour = hour;
30    repaint();                                                    repaint panel
31  }
32
33  /** Return minute */
34  public int getMinute() {
35    return minute;
36  }
37
38  /** Set a new minute */
39  public void setMinute(int minute) {
40    this.minute = minute;
41    repaint();                                                    repaint panel
42  }
43
44  /** Return second */
45  public int getSecond() {
46    return second;
47  }
48
49  /** Set a new second */
50  public void setSecond(int second) {
51    this.second = second;
52    repaint();                                                    repaint panel
53  }
54
55  /** Draw the clock */
56  protected void paintComponent(Graphics g) {                     override paintComponent
57    super.paintComponent(g);
58
59    // Initialize clock parameters
60    int clockRadius =
61      (int)(Math.min(getWidth(), getHeight()) * 0.8 * 0.5);
62    int xCenter = getWidth() / 2;
63    int yCenter = getHeight() / 2;
64
65    // Draw circle
66    g.setColor(Color.BLACK);
67    g.drawOval(xCenter - clockRadius, yCenter - clockRadius,
68      2 * clockRadius, 2 * clockRadius);
69    g.drawString("12", xCenter - 5, yCenter - clockRadius + 12);
70    g.drawString("9", xCenter - clockRadius + 3, yCenter + 5);
71    g.drawString("3", xCenter + clockRadius - 10, yCenter + 3);
72    g.drawString("6", xCenter - 3, yCenter + clockRadius - 3);
73
74    // Draw second hand
75    int sLength = (int)(clockRadius * 0.8);
```

```
76      int xSecond = (int)(xCenter + sLength *
77        Math.sin(second * (2 * Math.PI / 60)));
78      int ySecond = (int)(yCenter - sLength *
79        Math.cos(second * (2 * Math.PI / 60)));
80      g.setColor(Color.red);
81      g.drawLine(xCenter, yCenter, xSecond, ySecond);
82
83      // Draw minute hand
84      int mLength = (int)(clockRadius * 0.65);
85      int xMinute = (int)(xCenter + mLength *
86        Math.sin(minute * (2 * Math.PI / 60)));
87      int yMinute = (int)(yCenter - mLength *
88        Math.cos(minute * (2 * Math.PI / 60)));
89      g.setColor(Color.blue);
90      g.drawLine(xCenter, yCenter, xMinute, yMinute);
91
92      // Draw hour hand
93      int hLength = (int)(clockRadius * 0.5);
94      int xHour = (int)(xCenter + hLength *
95        Math.sin((hour % 12 + minute / 60.0) * (2 * Math.PI / 12)));
96      int yHour = (int)(yCenter - hLength *
97        Math.cos((hour % 12 + minute / 60.0) * (2 * Math.PI / 12)));
98      g.setColor(Color.green);
99      g.drawLine(xCenter, yCenter, xHour, yHour);
100   }
101
102   public void setCurrentTime() {
103     // Construct a calendar for the current date and time
104     Calendar calendar = new GregorianCalendar();
105
106     // Set current hour, minute and second
107     this.hour = calendar.get(Calendar.HOUR_OF_DAY);
108     this.minute = calendar.get(Calendar.MINUTE);
109     this.second = calendar.get(Calendar.SECOND);
110   }
111
112   public Dimension getPreferredSize() {
113     return new Dimension(200, 200);
114   }
115 }
```

get current time

override **getPreferredSize**

The program enables the clock size to adjust as the frame resizes. Every time you resize the frame, the **paintComponent** method is automatically invoked to paint the new frame. The **paintComponent** method displays the clock in proportion to the panel width (**getWidth()**) and height (**getHeight()**) (lines 60–63 in **StillClock**).

15.11 Displaying Images

You learned how to create image icons and display them in labels and buttons in §12.10, "Image Icons." For example, the following statements create an image icon and display it in a label:

```
ImageIcon icon = new ImageIcon("image/us.gif");
JLabel jlblImage = new JLabel(imageIcon);
```

An image icon displays a fixed-size image. To display an image in a flexible size, you need to use the **java.awt.Image** class. An image can be created from an image icon using the **getImage()** method as follows:

```
Image image = imageIcon.getImage();
```

Using a label as an area for displaying images is simple and convenient, but you don't have much control over how the image is displayed. A more flexible way to display images is to use the `drawImage` method of the `Graphics` class on a panel. Four versions of the `drawImage` method are shown in Figure 15.23.

java.awt.Graphics	
`+drawImage(image: Image, x: int, y: int, bgcolor: Color, observer: ImageObserver): void`	Draws the image in a specified location. The image's top-left corner is at (x, y) in the graphics context's coordinate space. Transparent pixels in the image are drawn in the specified color `bgcolor`. The observer is the object on which the image is displayed. The image is cut off if it is larger than the area it is being drawn on.
`+drawImage(image: Image, x: int, y: int, observer: ImageObserver): void`	Same as the preceding method except that it does not specify a background color.
`+drawImage(image: Image, x: int, y: int, width: int, height: int, observer: ImageObserver): void`	Draws a scaled version of the image that can fill all of the available space in the specified rectangle.
`+drawImage(image: Image, x: int, y: int, width: int, height: int, bgcolor: Color, observer: ImageObserver): void`	Same as the preceding method except that it provides a solid background color behind the image being drawn.

FIGURE 15.23 You can apply the `drawImage` method on a `Graphics` object to display an image in a GUI component.

`ImageObserver` specifies a GUI component for receiving notifications of image information as the image is constructed. To draw images using the `drawImage` method in a Swing component, such as `JPanel`, override the `paintComponent` method to tell the component how to display the image in the panel.

Listing 15.11 gives the code that displays an image from `image/us.gif`. The file `image/us.gif` (line 20) is under the class directory. An `Image` object is obtained in line 21. The `drawImage` method displays the image to fill in the whole panel, as shown in Figure 15.24.

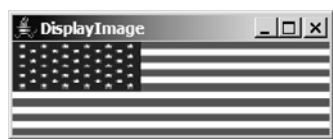


FIGURE 15.24 An image is displayed in a panel.

LISTING 15.11 DisplayImage.java

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class DisplayImage extends JFrame {
5    public DisplayImage() {
6      add(new ImagePanel());                                    add panel
7    }
8
9    public static void main(String[] args) {
10     JFrame frame = new DisplayImage();
11     frame.setTitle("DisplayImage");
12     frame.setSize(300, 300);
13     frame.setLocationRelativeTo(null); // Center the frame
14     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
15     frame.setVisible(true);
16   }
17 }
18
19 class ImagePanel extends JPanel {
20   private ImageIcon imageIcon = new ImageIcon("image/us.gif");
21   private Image image = imageIcon.getImage();
22
23   /** Draw image on the panel */
24   protected void paintComponent(Graphics g) {
25     super.paintComponent(g);
26
27     if (image != null)
28       g.drawImage(image, 0, 0, getWidth(), getHeight(), this);
29   }
30 }
```

panel class
create image icon
get image
override **paintComponent**
draw image

15.12 Case Study: The `ImageViewer` Class

Displaying an image is a common task in Java programming. This case study develops a reusable component named `ImageViewer` that displays an image on a panel. The class contains the properties `image`, `stretched`, `xCoordinate`, and `yCoordinate`, with associated accessor and mutator methods, as shown in Figure 15.25.

stretchable image

You can use images in Swing components such as `JLabel` and `JButton`, but these images are not stretchable. The image in an `ImageViewer` can be stretched.

Let us write a test program in Listing 15.12 that displays six images using the `ImageViewer` class. Figure 15.26 shows a sample run of the program.

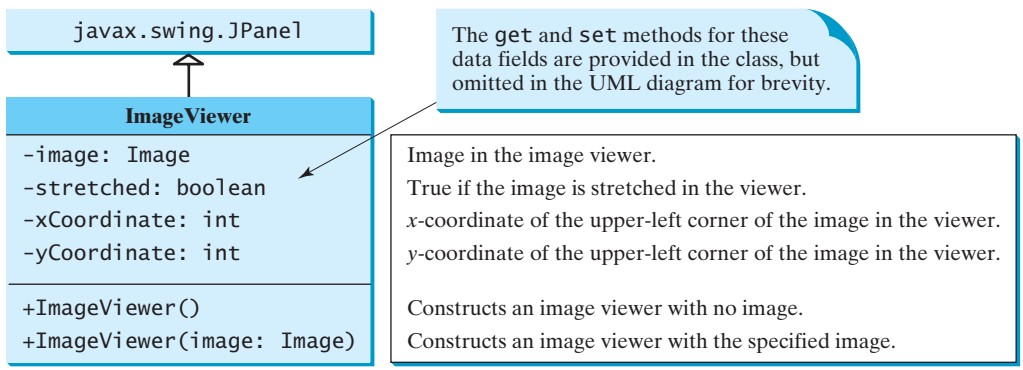

javax.swing.JPanel
ImageViewer
-image: Image
-stretched: boolean
-xCoordinate: int
-yCoordinate: int
+ImageViewer()
+ImageViewer(image: Image)
The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
Image in the image viewer.
True if the image is stretched in the viewer.
x-coordinate of the upper-left corner of the image in the viewer.
y-coordinate of the upper-left corner of the image in the viewer.
Constructs an image viewer with no image.
Constructs an image viewer with the specified image.

FIGURE 15.25 The `ImageViewer` class displays an image on a panel.



FIGURE 15.26 Six images are displayed in six `ImageViewer` components.

LISTING 15.12 SixFlags.java

```
1 import javax.swing.*;
2 import java.awt.*;
3
4 public class SixFlags extends JFrame {
5   public SixFlags() {
6     Image image1 = new ImageIcon("image/us.gif").getImage();                create image
7     Image image2 = new ImageIcon("image/ca.gif").getImage();
8     Image image3 = new ImageIcon("image/india.gif").getImage();
9     Image image4 = new ImageIcon("image/uk.gif").getImage();
10    Image image5 = new ImageIcon("image/china.gif").getImage();
11    Image image6 = new ImageIcon("image/norway.gif").getImage();
12
13    setLayout(new GridLayout(2, 0, 5, 5));
14    add(new ImageViewer(image1));                                           create image viewer
15    add(new ImageViewer(image2));
16    add(new ImageViewer(image3));
17    add(new ImageViewer(image4));
18    add(new ImageViewer(image5));
19    add(new ImageViewer(image6));
20  }
21
22  public static void main(String[] args) {
23    SixFlags frame = new SixFlags();
24    frame.setTitle("SixFlags");
25    frame.setSize(400, 320);
26    frame.setLocationRelativeTo(null); // Center the frame
27    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28    frame.setVisible(true);
29  }
30 }
```

implementation

skip implementation?

The **ImageViewer** class is implemented in Listing 15.13. (*Note: You may skip the implementation.*) The accessor and mutator methods for the properties **image**, **stretched**, **xCoordinate**, and **yCoordinate** are easy to implement. The **paintComponent** method (lines 26–35) displays the image on the panel. Line 29 ensures that the image is not **null** before displaying it. Line 30 checks whether the image is stretched or not.

LISTING 15.13 ImageViewer.java

```
1 import java.awt.*;
2 import javax.swing.*;
3
4 public class ImageViewer extends JPanel {
5   /** Hold value of property image. */
6   private java.awt.Image image;                                             properties
7
8   /** Hold value of property stretched. */
9   private boolean stretched = true;
10
11  /** Hold value of property xCoordinate. */
12  private int xCoordinate;
13
14  /** Hold value of property yCoordinate. */
15  private int yCoordinate;
16
```

constructor

```
17  /** Construct an empty image viewer */
18  public ImageViewer() {
19  }
```

constructor

```
20
21  /** Construct an image viewer for a specified Image object */
22  public ImageViewer(Image image) {
23    this.image = image;
24  }
25
26  protected void paintComponent(Graphics g) {
27    super.paintComponent(g);
28
```

image null?

```
29    if (image != null)
```

stretched

```
30      if (isStretched())
31        g.drawImage(image, xCoordinate, yCoordinate,
32          getWidth(), getHeight(), this);
```

nonstretched

```
33      else
34        g.drawImage(image, xCoordinate, yCoordinate, this);
35  }
36
37  /** Return value of property image */
38  public java.awt.Image getImage() {
39    return image;
40  }
41
42  /** Set a new value for property image */
43  public void setImage(java.awt.Image image) {
44    this.image = image;
45    repaint();
46  }
47
48  /** Return value of property stretched */
49  public boolean isStretched() {
50    return stretched;
51  }
52
53  /** Set a new value for property stretched */
54  public void setStretched(boolean stretched) {
55    this.stretched = stretched;
56    repaint();
57  }
58
59  /** Return value of property xCoordinate */
60  public int getXCoordinate() {
61    return xCoordinate;
62  }
63
64  /** Set a new value for property xCoordinate */
65  public void setXCoordinate(int xCoordinate) {
66    this.xCoordinate = xCoordinate;
67    repaint();
68  }
69
70  /** Return value of property yCoordinate */
71  public int getYCoordinate() {
72    return yCoordinate;
73  }
74
75  /** Set a new value for property yCoordinate */
```

```
76   public void setYCoordinate(int yCoordinate) {
77     this.yCoordinate = yCoordinate;
78     repaint();
79   }
80 }
```

Chapter Summary

1. Each component has its own coordinate system with the origin (`0, 0`) at the upper-left corner of the window. The *x*-coordinate increases to the right, and the *y*-coordinate increases downward.

2. The `Graphics` class is an abstract class for displaying figures and images on the screen on different platforms. The `Graphics` class is implemented on the native platform in the JVM. When you use the `paintComponent(g)` method to paint on a GUI component, this `g` is an instance of a concrete subclass of the abstract `Graphics` class for the specific platform. The `Graphics` class encapsulates the platform details and enables you to draw things uniformly without concern for the specific platform.

3. Invoking `super.paintComponent(g)` is necessary to ensure that the viewing area is cleared before a new drawing is displayed. The user can request the component to be redisplayed by invoking the `repaint()` method defined in the `Component` class. Invoking `repaint()` causes `paintComponent` to be invoked by the JVM. The user should never invoke `paintComponent` directly. For this reason, the protected visibility is sufficient for `paintComponent`.

4. Normally you use `JPanel` as a canvas. To draw on a `JPanel`, you create a new class that extends `JPanel` and overrides the `paintComponent` method to tell the panel how to draw things.

5. You can set fonts for the components or subjects you draw, and use font metrics to measure font size. Fonts and font metrics are encapsulated in the classes `Font` and `FontMetrics`. `FontMetrics` can be used to compute the exact length and width of a string, which is helpful for measuring the size of a string in order to display it in the right position.

6. The `Component` class has the `setBackground`, `setForeground`, and `setFont` methods. These methods are used to set colors and fonts for the entire component. Suppose you want to draw several messages in a panel with different colors and fonts; you have to use the `setColor` and `setFont` methods in the `Graphics` class to set the color and font for the current drawing.

7. To display an image, first create an image icon. You can then use `ImageIcon`'s `getImage()` method to get an `Image` object for the image and draw the image using the `drawImage` method in the `java.awt.Graphics` class.

Review Questions

Sections 15.2–15.3

15.1 Suppose that you want to draw a new message below an existing message. Should the `x`-coordinate, `y`-coordinate, or both increase or decrease?

15.2 Why is the `Graphics` class abstract? How is a `Graphics` object created?

15.3 Describe the `paintComponent` method. Where is it defined? How is it invoked? Can it be directly invoked? How can a program cause this method to be invoked?

15.4 Why is the `paintComponent` method `protected`? What happens if you change it to `public` or `private` in a subclass? Why is `super.paintComponent(g)` invoked in line 21 in Listing 15.1 and in line 31 in Listing 15.3?

15.5 Can you draw things on any Swing GUI component? Why should you use a panel as a canvas for drawings rather than a label or a button?

Sections 15.4-15.7

15.6 Describe the methods for drawing strings, lines, rectangles, round-cornered rectangles, 3D rectangles, ovals, arcs, polygons, and polylines.

15.7 Describe the methods for filling rectangles, round-cornered rectangle, ovals, arcs, and polygons.

15.8 How do you `get` and `set` colors and fonts in a `Graphics` object?

15.9 Write a statement to draw the following shapes:

- Draw a thick line from (`10`, `10`) to (`70`, `30`). You can draw several lines next to each other to create the effect of one thick line.
- Draw/fill a rectangle of width `100` and height `50` with the upper-left corner at (`10`, `10`).
- Draw/fill a round-cornered rectangle with width `100`, height `200`, corner horizontal diameter `40`, and corner vertical diameter `20`.
- Draw/fill a circle with radius `30`.
- Draw/fill an oval with width `50` and height `100`.
- Draw the upper half of a circle with radius `50`.
- Draw/fill a polygon connecting the following points: (`20`, `40`), (`30`, `50`), (`40`, `90`), (`90`, `10`), (`10`, `30`).

Sections 15.8–15.10

15.10 How do you find the leading, ascent, descent, and height of a font? How do you find the exact length in pixels of a string in a `Graphics` object?

15.11 If `message` is not initialized in line 8 in Listing 15.8, MessagePanel.java, what will happen when you create a `MessagePanel` using its no-arg constructor?

15.12 The following program is supposed to display a message on a panel, but nothing is displayed. There are problems in lines 2 and 14. Correct them.

```
 1 public class TestDrawMessage extends javax.swing.JFrame {
 2   public void TestDrawMessage() {
 3     add(new DrawMessage());
 4   }
 5
 6   public static void main(String[] args) {
 7     javax.swing.JFrame frame = new TestDrawMessage();
 8     frame.setSize(100, 200);
 9     frame.setVisible(true);
10   }
11 }
12
13 class DrawMessage extends javax.swing.JPanel {
14   protected void PaintComponent(java.awt.Graphics g) {
15     super.paintComponent(g);
16     g.drawString("Welcome to Java", 20, 20);
17   }
18 }
```

Sections 15.11–15.12

15.13 How do you create an `Image` object from the `ImageIcon` object?

15.14 How do you create an `ImageIcon` object from an `Image` object?

15.15 Describe the `drawImage` method in the `Graphics` class.

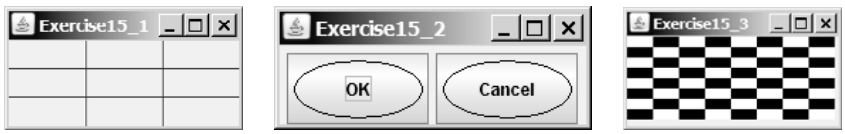
15.16 Explain the differences between displaying images in a `JLabel` and in a `JPanel`.

15.17 Which package contains `ImageIcon`, and which contains `Image`?

Programming Exercises

Sections 15.2–15.7

15.1* (*Displaying a 3 × 3 grid*) Write a program that displays a 3 × 3 grid, as shown in Figure 15.27(a). Use red color for vertical lines and blue for horizontals.



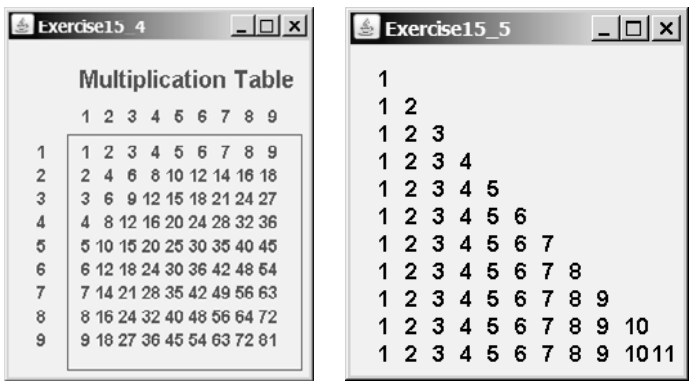
(a) (b) (c)

Figure 15.27 (a) Exercise 15.1 displays a grid. (b) Exercise 15.2 displays two objects of `OvalButton`. (c) Exercise 15.3 displays a checkerboard.

15.2** (*Creating a custom button class*) Develop a custom button class named `OvalButton` that extends `JButton` and displays the button text inside an oval. Figure 15.27(b) shows two buttons created using the `OvalButton` class.

15.3* (*Displaying a checkerboard*) Exercise 12.10 displays a checkerboard in which each white and black cell is a `JButton`. Rewrite a program that draws a checkerboard on a `JPanel` using the drawing methods in the `Graphics` class, as shown in Figure 15.27(c).

15.4* (*Displaying a multiplication table*) Write a program that displays a multiplication table in a panel using the drawing methods, as shown in Figure 15.28(a).



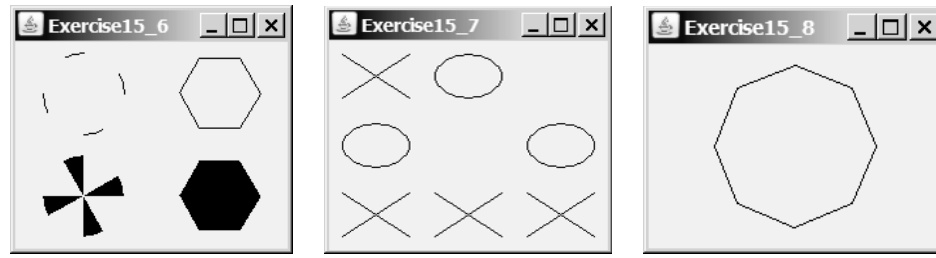
(a) (b)

Figure 15.28 (a) Exercise 15.4 displays a multiplication table. (b) Exercise 15.5 displays numbers in a triangle formation.

15.5** (*Displaying numbers in a triangular pattern*) Write a program that displays numbers in a triangular pattern, as shown in Figure 15.28(b). The number of lines in the display changes to fit the window as the window resizes.

15.6** (*Improving `FigurePanel`*) The `FigurePanel` class in Listing 15.3 can display lines, rectangles, round-cornered rectangles, and ovals. Add appropriate new code

in the class to display arcs and polygons. Write a test program to display the shapes as shown in Figure 15.29(a) using the new `FigurePanel` class.



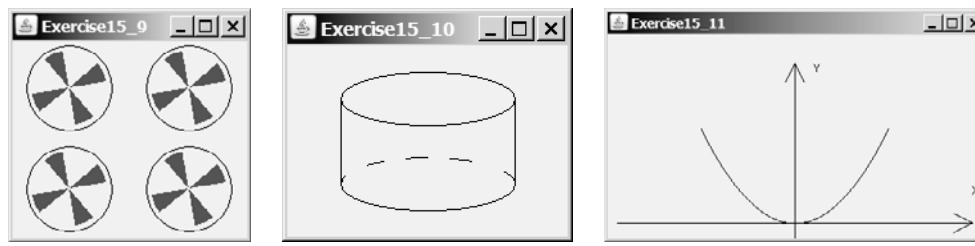
(a) (b) (c)

FIGURE 15.29 (a) Four panels of geometric figures are displayed in a frame of `GridLayout`. (b) `TicTacToe` cells randomly display X, O, or nothing. (c) Exercise 15.8 draws an octagon.

15.7** (*Displaying a TicTacToe board*) Create a custom panel that displays `X`, `O`, or nothing. What to display is randomly decided whenever a panel is repainted. Use the `Math.random()` method to generate an integer `0`, `1`, or `2`, which corresponds to displaying `X`, `O`, or nothing. Create a frame that contains nine custom panels, as shown in Figure 15.29(b).

15.8** (*Drawing an octagon*) Write a program that draws an octagon, as shown in Figure 15.29(c).

15.9* (*Creating four fans*) Write a program that places four fans in a frame of `GridLayout` with two rows and two columns, as shown in Figure 15.30(a).



(a) (b) (c)

FIGURE 15.30 (a) Exercise 15.9 draws four fans. (b) Exercise 15.10 draws a cylinder. (c) Exercise 15.11 draws a diagram for function $f(x) = x^2$.

15.10* (*Displaying a cylinder*) Write a program that draws a cylinder, as shown in Figure 15.30(b).

15.11** (*Plotting the square function*) Write a program that draws a diagram for the function $f(x) = x^2$ (see Figure 15.30(c)).

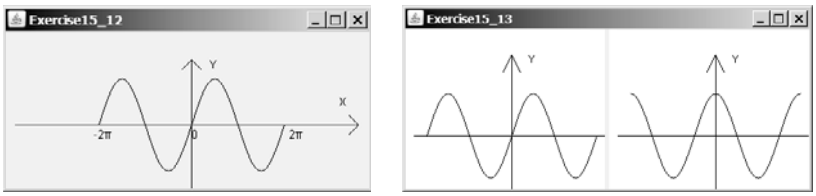
Hint: Add points to a polygon `p` using the following loop:

```
double scaleFactor = 0.1;

for (int x = -100; x <= 100; x++) {
  p.addPoint(x + 200, 200 - (int)(scaleFactor * x * x));
}
```

Connect the points using `g.drawPolyline(p.xpoints, p.ypoints, p.npoints)` for a `Graphics` object `g`. `p.xpoints` returns an array of `x`-coordinates, `p.ypoints` an array of `y`-coordinates, and `p.npoints` the number of points in `Polygon` object `p`.

15.12** (*Plotting the sine function*) Write a program that draws a diagram for the sine function, as shown in Figure 15.31(a).



(a) (b)

FIGURE 15.31 (a) Exercise 15.12 draws a diagram for function $f(x) = sin(x)$. (b) Exercise 15.13 draws the sine and cosine functions.

Hint: The Unicode for π is `\u03c0`. To display -2π, use `g.drawString("-2\u03c0", x, y)`. For a trigonometric function like `sin(x)`, `x` is in radians. Use the following loop to add the points to a polygon `p`:

```
for (int x = -100; x <= 100; x++) {
  p.addPoint(x + 200,
    100 - (int)(50 * Math.sin((x / 100.0) * 2 * Math.PI)));
}
```

-2π is at (100, 100), the center of the axis is at (200, 100), and 2π is at (300, 100). Use the `drawPolyline` method in the `Graphics` class to connect the points.

15.13** (*Plotting functions using abstract methods*) Write an abstract class that draws the diagram for a function. The class is defined as follows:

```
public abstract class AbstractDrawFunction extends JPanel {
  /** Polygon to hold the points */
  private Polygon p = new Polygon();

  protected AbstractDrawFunction () {
    drawFunction();
  }

  /** Return the y-coordinate */
  abstract double f(double x);

  /** Obtain points for x-coordinates 100, 101, ..., 300 */
  public void drawFunction() {
    for (int x = -100; x <= 100; x++) {
      p.addPoint(x + 200, 200 - (int)f(x));
    }
  }

  /** Implement paintComponent to draw axes, labels, and
   *  connecting points
   */
  protected void paintComponent(Graphics g) {
    // To be completed by you
  }
}
```



Video Note
Plot a sine function

Test the class with the following functions:

```
f(x) = x^2;
f(x) = sin(x);
f(x) = cos(x);
f(x) = tan(x);
f(x) = cos(x) + 5sin(x);
f(x) = 5cos(x) + sin(x);
f(x) = log(x) + x^2;
```

For each function, create a class that extends the `AbstractDrawFunction` class and implements the `f` method. Figure 15.31(b) displays the drawings for the sine function and the cosine function.

15.14** (*Displaying a bar chart*) Write a program that uses a bar chart to display the percentages of the overall grade represented by projects, quizzes, midterm exams, and the final exam, as shown in Figure 15.1(a). Suppose that projects take 20 percent and are displayed in red, quizzes take 10 percent and are displayed in blue, midterm exams take 30 percent and are displayed in green, and the final exam takes 40 percent and is displayed in orange.

15.15** (*Displaying a pie chart*) Write a program that uses a pie chart to display the percentages of the overall grade represented by projects, quizzes, midterm exam, and the final exam, as shown in Figure 15.32(a). Suppose that projects take 20 percent and are displayed in red, quizzes take 10 percent and are displayed in blue, midterm exam takes 30 percent and are displayed in green, and the final exam takes 40 percent and is displayed in orange.

15.16 (*Obtaining font information*) Write a program that displays the message "Java is fun" in a panel. Set the panel's font to `TimesRoman`, `bold`, and 20 pixel. Display the font's leading, ascent, descent, height, and the string width as a tool tip text for the panel, as shown in Figure 15.32(b).

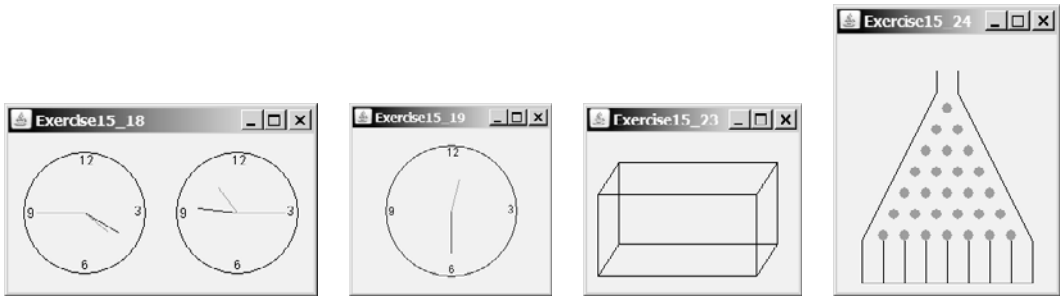


(a) (b) (c)

FIGURE 15.32 (a) Exercise 15.15 uses a pie chart to show the percentages of projects, quizzes, midterm exam, and final exam in the overall grade. (b) Exercise 15.16 displays font properties in a tool tip text. (c) Exercise 15.17 draws a sketch for the hangman game.

15.17 (*Game: hangman*) Write a program that displays a drawing for the popular hangman game, as shown in Figure 15.32(c).

15.18 (*Using the `StillClock` class*) Write a program that displays two clocks. The hour, minute, and second values are 4, 20, 45 for the first clock and 22, 46, 15 for the second clock, as shown in Figure 15.33(a).



(a) (b) (c) (d)

FIGURE 15.33 (a) Exercise 15.18 displays two clocks. (b) Exercise 15.19 displays a clock with random hour and minute values. (c) Exercise 15.23 displays a rectanguloid. (d) Exercise 15.24 simulates a bean machine.

15.19* (*Random time*) Modify the `StillClock` class with three new Boolean properties—`hourHandVisible`, `minuteHandVisible`, and `secondHandVisible`—and their associated accessor and mutator methods. You can use the `set` methods to make a hand visible or invisible. Write a test program that displays only the hour and minute hands. The hour and minute values are randomly generated. The hour is between `0` and `11`, and the minute is either `0` or `30`, as shown in Figure 15.33(b).

15.20** (*Drawing a detailed clock*) Modify the `StillClock` class in §15.12, "Case Study: The `StillClock` Class," to draw the clock with more details on the hours and minutes, as shown in Figure 15.1(b).

15.21** (*Displaying a TicTacToe board with images*) Rewrite Exercise 12.7 to display an image in a `JPanel` instead of displaying an image icon in a `JLabel`.

15.22** (*Displaying a STOP sign*) Write a program that displays a STOP sign, as shown in Figure 15.1(c). The hexagon is in red and the sign is in white.

(*Hint*: See Listing 15.5, DrawPolygon.java, and Listing 15.6, TestCenterMessage.java.)

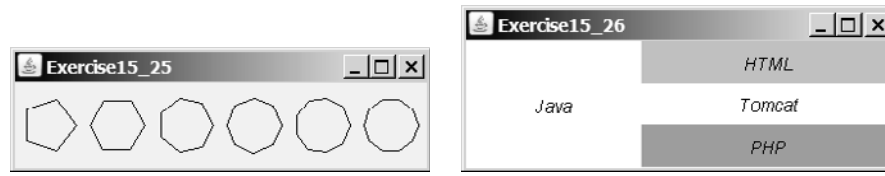
15.23 (*Displaying a rectanguloid*) Write a program that displays a rectanguloid, as shown in Figure 15.33(c). The cube should grow and shrink as the frame grows or shrinks.

15.24** (*Game: bean machine*) Write a program that displays a bean machine introduced in Exercise 6.21. The bean machine should be centered in a resizable panel, as shown in Figure 15.33(d).

15.25** (*Geometry: displaying an n-sided regular polygon*) Create a subclass of `JPanel`, named `RegularPolygonPanel`, to paint an *n*-sided regular polygon. The class contains a property named `numberOfSides`, which specifies the number of sides in the polygon. The polygon is centered at the center of the panel. The size of the polygon is proportional to the size of the panel. Create a pentagon, hexagon, heptagon, and octagon, nonagon, and decagon from `RegularPolygonPanel` and display them in a frame, as shown in Figure 15.34(a).

15.26 (*Using the `MessagePanel` class*) Write a program that displays four messages, as shown in Figure 15.34(b).

15.27** (*Displaying a graph*) A graph consists of vertices and edges that connect vertices. Write a program that reads a graph from a file and displays it on a panel. The first line in the file contains a number that indicates the number of vertices (`n`). The vertices are labeled as `0`, `1`, ..., `n-1`. Each subsequent line, with the

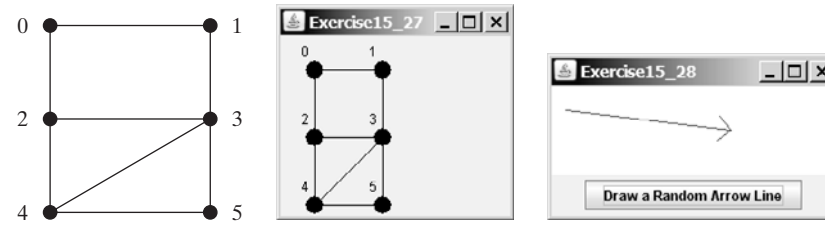


(a) (b)

FIGURE 15.34 (a) Exercise 15.25 displays several n-sided polygons. (b) Exercise 15.26 uses `MessagePanel` to display four strings.

format `u x y v1, v2, ...`, describes that the vertex `u` is located at position (`x`, `y`) with edges (`u`, `v1`), (`u`, `v2`), etc. Figure 15.35(a) gives an example of the file for a graph. Your program prompts the user to enter the name of the file, reads data from the file, and displays the graph on a panel, as shown in Figure 15.35(b).

```
File
6
0  30  30  1  2
1  90  30  0  3
2  30  90  0  3  4
3  90  90  1  2  4  5
4  30  150  2  3  5
5  90  150  3  4
```



(a) (b) (c)

FIGURE 15.35 (a)-(b) The program reads the information about the graph and displays it visually. (c) The program displays an arrow line.

15.28** (*Drawing an arrow line*) Write a static method that draws an arrow line from a starting point to an ending point using the following method header:

```
public static void drawArrowLine(int x1, int y1,
  int x2, int y2, Graphics g)
```

Write a test program that randomly draws an arrow line when the *Draw Random Arrow Line* button is clicked, as shown in Figure 15.35(c).

CHAPTER 16

EVENT-DRIVEN PROGRAMMING

Objectives

- To describe events, event sources, and event classes (§16.2).
- To define listener classes, register listener objects with the source object, and write the code to handle events (§16.3).
- To define listener classes using inner classes (§16.4).
- To define listener classes using anonymous inner classes (§16.5).
- To explore various coding styles for creating and registering listeners (§16.6).
- To get input from text field upon clicking a button (§16.7).
- To write programs to deal with `WindowEvent` (§16.8).
- To simplify coding for listener classes using listener interface adapters (§16.9).
- To write programs to deal with `MouseEvent` (§16.10).
- To write programs to deal with `KeyEvent` (§16.11).
- To use the `javax.swing.Timer` class to control animations (§16.12).



16.1 Introduction

problem

Suppose you wish to write a GUI program that lets the user enter the loan amount, annual interest rate, and number of years and click the *Compute Loan* button to obtain the monthly payment and total payment, as shown in Figure 16.1(a). How do you accomplish the task? You have to use event-driven programming to write the code to respond to the button-clicking event.



(a) (b) (c) (d)

FIGURE 16.1 (a) The program computes loan payments. (b)–(d) A flag is rising upward.

problem

Suppose you wish to write a program that animates a rising flag, as shown in Figure 16.1(b)–(d). How do you accomplish the task? There are several ways to solve this problem. An effective one is to use a timer in event-driven programming, which is the subject of this chapter.

In event-driven programming, code is executed when an event occurs (e.g., a button click, a mouse movement, or a timer). §14.6, "Example: The `ActionListener` Interface," gave you a taste of event-driven programming. You probably have many questions, such as why a listener class is defined to implement the `ActionListener` interface. This chapter will give you all the answers.

16.2 Event and Event Source

event

When you run a Java GUI program, the program interacts with the user, and the events drive its execution. An *event* can be defined as a signal to the program that something has happened. Events are triggered either by external user actions, such as mouse movements, button clicks, and keystrokes, or by internal program activities, such as a timer. The program can choose to respond to or ignore an event.

fire event
source object

The component that creates an event and fires it is called the *source object* or *source component*. For example, a button is the source object for a button-clicking action event. An event is an instance of an event class. The root class of the event classes is `java.util.EventObject`. The hierarchical relationships of some event classes are shown in Figure 16.2.

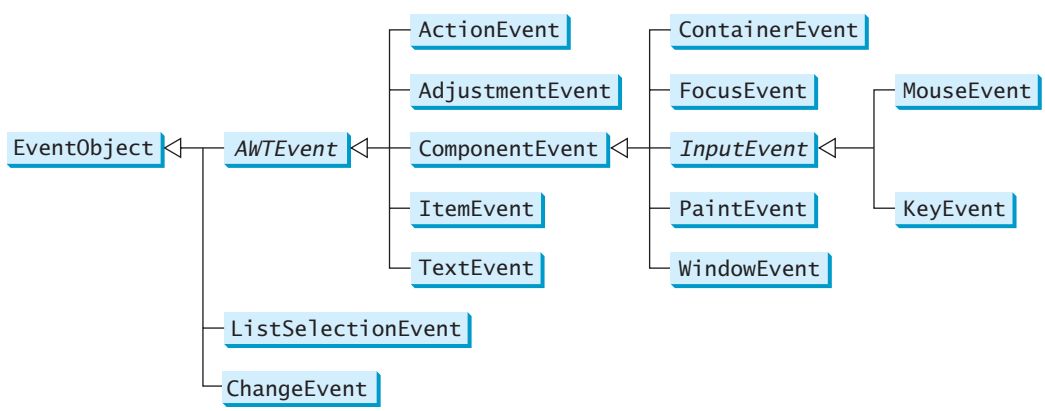


FIGURE 16.2 An event is an object of the `EventObject` class.

An event object contains whatever properties are pertinent to the event. You can identify the source object of an event using the `getSource()` instance method in the `EventObject` class. The subclasses of `EventObject` deal with special types of events, such as action events, window events, component events, mouse events, and key events. Table 16.1 lists external user actions, source objects, and event types fired.

getSource()

TABLE 16.1 User Action, Source Object, and Event Type

User Action	*Source Object*	*Event Type Fired*
Click a button	`JButton`	`ActionEvent`
Press return on a text field	`JTextField`	`ActionEvent`
Select a new item	`JComboBox`	`ItemEvent`, `ActionEvent`
Select item(s)	`JList`	`ListSelectionEvent`
Click a check box	`JCheckBox`	`ItemEvent`, `ActionEvent`
Click a radio button	`JRadioButton`	`ItemEvent`, `ActionEvent`
Select a menu item	`JMenuItem`	`ActionEvent`
Move the scroll bar	`JScrollBar`	`AdjustmentEvent`
Move the scroll bar	`JSlider`	`ChangeEvent`
Window opened, closed, iconified, deiconified, or closing	`Window`	`WindowEvent`
Mouse pressed, released, clicked, entered, or exited	`Component`	`MouseEvent`
Mouse moved or dragged	`Component`	`MouseEvent`
Key released or pressed	`Component`	`KeyEvent`
Component added or removed from the container	`Container`	`ContainerEvent`
Component moved, resized, hidden, or shown	`Component`	`ComponentEvent`
Component gained or lost focus	`Component`	`FocusEvent`



Note

If a component can fire an event, any subclass of the component can fire the same type of event. For example, every GUI component can fire `MouseEvent`, `KeyEvent`, `FocusEvent`, and `ComponentEvent`, since `Component` is the superclass of all GUI components.



Note

All the event classes in Figure 16.2 are included in the `java.awt.event` package except `ListSelectionEvent` and `ChangeEvent`, which are in the `javax.swing.event` package. AWT events were originally designed for AWT components, but many Swing components fire them.

16.3 Listeners, Registrations, and Handling Events

Java uses a delegation-based model for event handling: a source object fires an event, and an object interested in the event handles it. The latter object is called a *listener*. For an object to be a listener for an event on a source object, two things are needed, as shown in Figure 16.3.

listener
ActionEvent/ActionListener

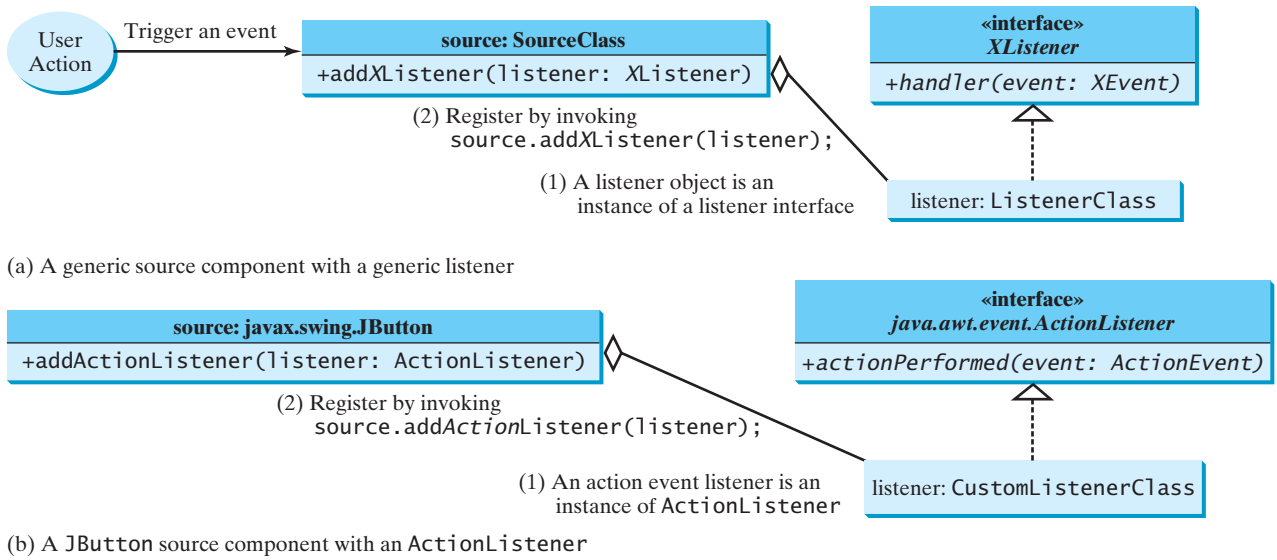


FIGURE 16.3 A listener must be an instance of a listener interface and must be registered with a source component.

listener interface
*X*Listener/*X*Event

1. The listener object must be an instance of the corresponding event-listener interface to ensure that the listener has the correct method for processing the event. Java provides a listener interface for every type of event. The listener interface is usually named **`XListener`** for **`XEvent`**, with the exception of **`MouseMotionListener`**. For example, the corresponding listener interface for **`ActionEvent`** is **`ActionListener`**; each listener for **`ActionEvent`** should implement the **`ActionListener`** interface. Table 16.2 lists event types, the corresponding listener interfaces, and the methods defined in the listener interfaces. The listener interface contains the method(s), known as the *handler(s)*, for processing the event.

handler

register listener

2. The listener object must be registered by the source object. Registration methods depend on the event type. For **`ActionEvent`**, the method is **`addActionListener`**. In general, the method is named **`addXListener`** for **`XEvent`**. A source object may fire several types of events. It maintains, for each event, a list of registered listeners and notifies them by invoking the *handler* of the listener object to respond to the event, as shown in Figure 16.4. (Figure 16.4 shows the internal implementation of a source class. You don't have to know how a source class such as **`JButton`** is implemented in order to use it. Nevertheless, this knowledge will help you to understand the Java event-driven programming framework).

Let's revisit Listing 14.8, HandleEvent.java. Since a **`JButton`** object fires **`ActionEvent`**, a listener object for **`ActionEvent`** must be an instance of **`ActionListener`**, so the listener class implements **`ActionListener`** in line 34. The source object invokes **`addActionListener(listener)`** to register a listener, as follows:

create source object
create listener object
register listener

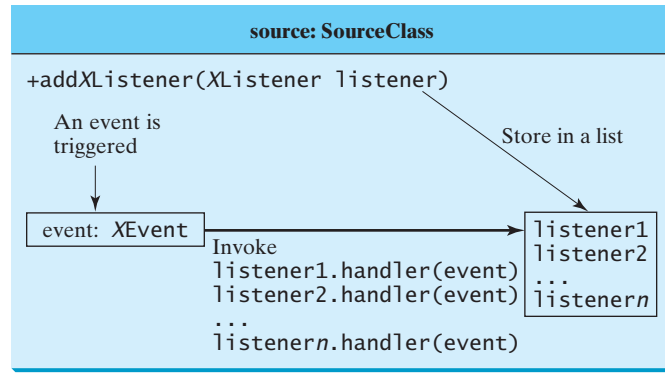
```
JButton jbtOK = new JButton("OK"); // Line 7 in Listing 14.8
ActionListener listener1
   = new OKListenerClass(); // Line 18 in Listing 14.8
jbtOK.addActionListener(listener1); // Line 20 in Listing 14.8
```

When you click the button, the **`JButton`** object fires an **`ActionEvent`** and passes it to invoke the listener's **`actionPerformed`** method to handle the event.

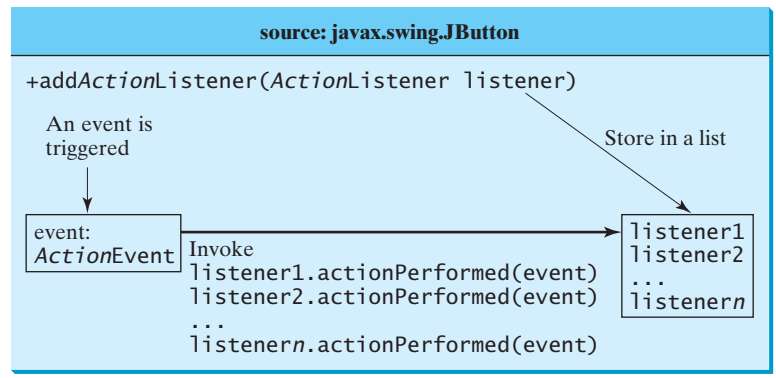
TABLE 16.2 Events, Event Listeners, and Listener Methods

Event Class (Handlers)	*Listener Interface*	*Listener Methods*
`ActionEvent`	`ActionListener`	`actionPerformed(ActionEvent)`
`ItemEvent`	`ItemListener`	`itemStateChanged(ItemEvent)`
`MouseEvent`	`MouseListener`	`mousePressed(MouseEvent)`
		`mouseReleased(MouseEvent)`
		`mouseEntered(MouseEvent)`
		`mouseExited(MouseEvent)`
		`mouseClicked(MouseEvent)`
	`MouseMotionListener`	`mouseDragged(MouseEvent)`
		`mouseMoved(MouseEvent)`
`KeyEvent`	`KeyListener`	`keyPressed(KeyEvent)`
		`keyReleased(KeyEvent)`
		`keyTyped(KeyEvent)`
`WindowEvent`	`WindowListener`	`windowClosing(WindowEvent)`
		`windowOpened(WindowEvent)`
		`windowIconified(WindowEvent)`
		`windowDeiconified(WindowEvent)`
		`windowClosed(WindowEvent)`
		`windowActivated(WindowEvent)`
		`windowDeactivated(WindowEvent)`
`ContainerEvent`	`ContainerListener`	`componentAdded(ContainerEvent)`
		`componentRemoved(ContainerEvent)`
`ComponentEvent`	`ComponentListener`	`componentMoved(ComponentEvent)`
		`componentHidden(ComponentEvent)`
		`componentResized(ComponentEvent)`
		`componentShown(ComponentEvent)`
`FocusEvent`	`FocusListener`	`focusGained(FocusEvent)`
		`focusLost(FocusEvent)`
`AdjustmentEvent`	`AdjustmentListener`	`adjustmentValueChanged(AdjustmentEvent)`
`ChangeEvent`	`ChangeListener`	`stateChanged(ChangeEvent)`
`ListSelectionEvent`	`ListSelectionListener`	`valueChanged(ListSelectionEvent)`

The event object contains information pertinent to the event, which can be obtained using the methods, as shown in Figure 16.5. For example, you can use `e.getSource()` to obtain the source object in order to determine whether it is a button, a check box, or a radio button. For an action event, you can use `e.getWhen()` to obtain the time when the event occurs.



(a) Internal function of a generic source object



(b) Internal function of a `JButton` object

FIGURE 16.4 The source object notifies the listeners of the event by invoking the handler of the listener object.

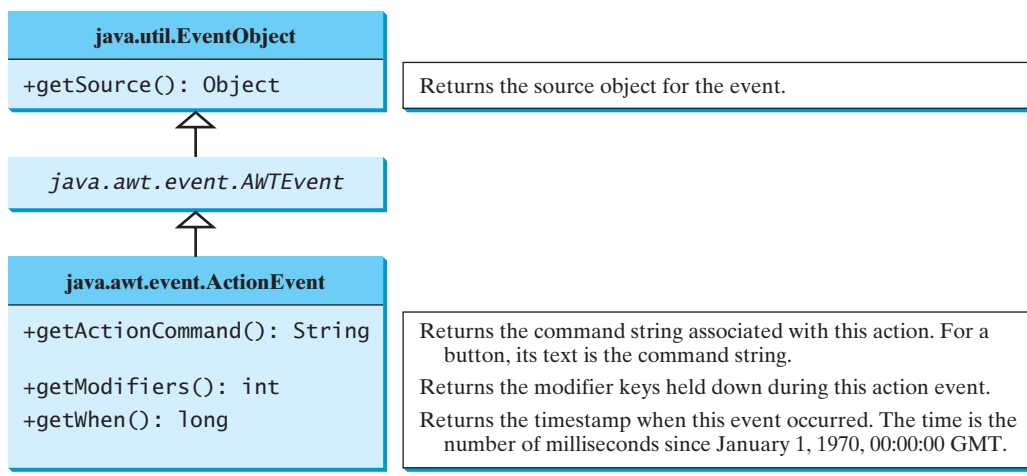

java.util.EventObject
+getSource(): Object
Returns the source object for the event.
java.awt.event.AWTEvent
java.awt.event.ActionEvent
+getActionCommand(): String
+getModifiers(): int
+getWhen(): long
Returns the command string associated with this action. For a button, its text is the command string.
Returns the modifier keys held down during this action event.
Returns the timestamp when this event occurred. The time is the number of milliseconds since January 1, 1970, 00:00:00 GMT.

FIGURE 16.5 You can obtain useful information from an event object.

We now write a program that uses two buttons to control the size of a circle, as shown in Figure 16.6.

first version

We will develop this program incrementally. First we write a program in Listing 16.1 that displays the user interface with a circle in the center (line 14) and two buttons in the bottom (line 15).



FIGURE 16.6 The user clicks the *Enlarge* and *Shrink* buttons to enlarge and shrink the size of the circle.

LISTING 16.1 ControlCircle1.java

```
1  import javax.swing.*;
2  import java.awt.*;
3
4  public class ControlCircle1 extends JFrame {
5    private JButton jbtEnlarge = new JButton("Enlarge");                    buttons
6    private JButton jbtShrink = new JButton("Shrink");
7    private CirclePanel canvas = new CirclePanel();                         circle canvas
8
9    public ControlCircle1() {
10     JPanel panel = new JPanel(); // Use the panel to group buttons
11     panel.add(jbtEnlarge);
12     panel.add(jbtShrink);
13
14     this.add(canvas, BorderLayout.CENTER); // Add canvas to center
15     this.add(panel, BorderLayout.SOUTH); // Add buttons to the frame
16   }
17
18   /** Main method */
19   public static void main(String[] args) {
20     JFrame frame = new ControlCircle1();
21     frame.setTitle("ControlCircle1");
22     frame.setLocationRelativeTo(null); // Center the frame
23     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24     frame.setSize(200, 200);
25     frame.setVisible(true);
26   }
27 }
28
29 class CirclePanel extends JPanel {                                         CirclePanel class
30   private int radius = 5; // Default circle radius
31
32   /** Repaint the circle */
33   protected void paintComponent(Graphics g) {                              paint the circle
34     super.paintComponent(g);
35     g.drawOval(getWidth() / 2 - radius, getHeight() / 2 - radius,
36       2 * radius, 2 * radius);
37   }
38 }
```

How do you use the buttons to enlarge or shrink the circle? When the *Enlarge* button is clicked, you want the circle to be repainted with a larger radius. How can you accomplish this? You can expand the program in Listing 16.1 into Listing 16.2 with the following features: second version

1. Define a listener class named `EnlargeListener` that implements `ActionListener` (lines 31–35).
2. Create a listener and register it with `jbtEnlarge` (line 18).

3. Add a method named `enlarge()` in `CirclePanel` to increase the radius, then repaint the panel (lines 41–44).

4. Implement the `actionPerformed` method in `EnlargeListener` to invoke `canvas.enlarge()` (line 33).

5. To make the reference variable `canvas` accessible from the `actionPerformed` method, define `EnlargeListener` as an inner class of the `ControlCircle2` class (lines 31–35). Inner classes are defined inside another class. We will introduce inner classes in the next section.

6. To avoid compile errors, the `CirclePanel` class (lines 37–52) now is also defined as an inner class in `ControlCircle2`,since an old `CirclePanel` class is already defined in Listing 16.1.

LISTING 16.2 ControlCircle2.java



Video Note
Listener and its registration

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4
5  public class ControlCircle2 extends JFrame {
6    private JButton jbtEnlarge = new JButton("Enlarge");
7    private JButton jbtShrink = new JButton("Shrink");
8    private CirclePanel canvas = new CirclePanel();
9
10   public ControlCircle2() {
11     JPanel panel = new JPanel(); // Use the panel to group buttons
12     panel.add(jbtEnlarge);
13     panel.add(jbtShrink);
14
15     this.add(canvas, BorderLayout.CENTER); // Add canvas to center
16     this.add(panel, BorderLayout.SOUTH); // Add buttons to the frame
17
18     jbtEnlarge.addActionListener(new EnlargeListener());
19   }
20
21   /** Main method */
22   public static void main(String[] args) {
23     JFrame frame = new ControlCircle2();
24     frame.setTitle("ControlCircle2");
25     frame.setLocationRelativeTo(null); // Center the frame
26     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27     frame.setSize(200, 200);
28     frame.setVisible(true);
29   }
30
31   class EnlargeListener implements ActionListener { // Inner class
32     public void actionPerformed(ActionEvent e) {
33       canvas.enlarge();
34     }
35   }
36
37   class CirclePanel extends JPanel { // Inner class
38     private int radius = 5; // Default circle radius
39
40     /** Enlarge the circle */
41     public void enlarge() {
42       radius++;
```

create/register listener

listener class

CirclePanel class

enlarge method

```
43        repaint();
44      }
45
46      /** Repaint the circle */
47      protected void paintComponent(Graphics g) {
48        super.paintComponent(g);
49        g.drawOval(getWidth() / 2 - radius, getHeight() / 2 - radius,
50          2 * radius, 2 * radius);
51      }
52    }
53 }
```

Similarly you can add the code for the *Shrink* button to display a smaller circle when the *Shrink* button is clicked.

the Shrink button

16.4 Inner Classes

An *inner class*, or *nested class,* is a class defined within the scope of another class. The code in Figure 16.7(a) defines two separate classes, `Test` and `A`. The code in Figure 16.7(b) defines `A` as an inner class in `Test`.

```
public class Test {
  ...
}

public class A {
  ...
}
```

(a)

```
public class Test {
  ...

  // Inner class
  public class A {
  ...
  }
}
```

(b)

```
// OuterClass.java: inner class demo
public class OuterClass {
  private int data;

  /** A method in the outer class */
  public void m() {
    // Do something
  }

  // An inner class
  class InnerClass {
    /** A method in the inner class */
    public void mi() {
      // Directly reference data and method
      // defined in its outer class
      data++;
      m();
    }
  }
}
```

(c)

FIGURE 16.7 Inner classes combine dependent classes into the primary class.

The class `InnerClass` defined inside `OuterClass` in Figure 16.7(c) is another example of an inner class. An inner class may be used just like a regular class. Normally, you define a class an inner class if it is used only by its outer class. An inner class has the following features:

- An inner class is compiled into a class named *`OuterClassName$InnerClassName.class`*. For example, the inner class `A` in `Test` is compiled into *`Test$A.class`* in Figure 16.7(b).
- An inner class can reference the data and methods defined in the outer class in which it nests, so you need not pass the reference of an object of the outer class to the constructor of the inner class. For this reason, inner classes can make programs simple and concise.
- An inner class can be defined with a visibility modifier subject to the same visibility rules applied to a member of the class.

- An inner class can be defined `static`. A `static` inner class can be accessed using the outer class name. A `static` inner class cannot access nonstatic members of the outer class.
- Objects of an inner class are often created in the outer class. But you can also create an object of an inner class from another class. If the inner class is nonstatic, you must first create an instance of the outer class, then use the following syntax to create an object for the inner class:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

- If the inner class is static, use the following syntax to create an object for it:

```
OuterClass.InnerClass innerObject = new OuterClass.InnerClass();
```

A simple use of inner classes is to combine dependent classes into a primary class. This reduces the number of source files. It also makes class files easy to organize, since they are all named with the primary class as the prefix. For example, rather than creating two source files, **Test.java** and **A.java**, in Figure 16.7(a), you can combine class `A` into class `Test` and create just one source file **Test.java** in Figure 16.7(b). The resulting class files are **Test.class** and **Test$A.class**.

Another practical use of inner classes is to avoid class-naming conflict. Two versions of `CirclePanel` are defined in Listings 16.1 and 16.2. You can define them as inner classes to avoid conflict.

16.5 Anonymous Class Listeners

A listener class is designed specifically to create a listener object for a GUI component (e.g., a button). The listener class will not be shared by other applications and therefore is appropriate to be defined inside the frame class as an inner class.

anonymous inner class

Inner-class listeners can be shortened using anonymous inner classes. An *anonymous inner class* is an inner class without a name. It combines defining an inner class and creating an instance of the class in one step. The inner class in Listing 16.2 can be replaced by an anonymous inner class as shown below.

```
public ControlCircle2() {
  // Omitted

  jbtEnlarge.addActionListener(
    new EnlargeListener());
}

class EnlargeListener
    implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    canvas.enlarge();
  }
}
```

(a) Inner class EnlargeListener

```
public ControlCircle2() {
  // Omitted

  jbtEnlarge.addActionListener(
    new class EnlargeListener
        implements ActionListener() {
      public void
          actionPerformed(ActionEvent e) {
        canvas.enlarge();
      }
    });
}
```

(b) Anonymous inner class

The syntax for an anonymous inner class is as follows:

```
new SuperClassName/InterfaceName() {
  // Implement or override methods in superclass or interface

  // Other methods if necessary
}
```

Since an anonymous inner class is a special kind of inner class, it is treated like an inner class with the following features:

- An anonymous inner class must always extend a superclass or implement an interface, but it cannot have an explicit `extends` or `implements` clause.
- An anonymous inner class must implement all the abstract methods in the superclass or in the interface.
- An anonymous inner class always uses the no-arg constructor from its superclass to create an instance. If an anonymous inner class implements an interface, the constructor is `Object()`.
- An anonymous inner class is compiled into a class named `OuterClassName$n.class`. For example, if the outer class `Test` has two anonymous inner classes, they are compiled into `Test$1.class` and `Test$2.class`.

Listing 16.3 gives an example that handles the events from four buttons, as shown in Figure 16.8.



FIGURE 16.8 The program handles the events from four buttons.

LISTING 16.3 AnonymousListenerDemo.java



Video Note
Anonymous listener

```
1  import javax.swing.*;
2  import java.awt.event.*;
3
4  public class AnonymousListenerDemo extends JFrame {
5    public AnonymousListenerDemo() {
6      // Create four buttons
7      JButton jbtNew = new JButton("New");
8      JButton jbtOpen = new JButton("Open");
9      JButton jbtSave = new JButton("Save");
10     JButton jbtPrint = new JButton("Print");
11
12     // Create a panel to hold buttons
13     JPanel panel = new JPanel();
14     panel.add(jbtNew);
15     panel.add(jbtOpen);
16     panel.add(jbtSave);
17     panel.add(jbtPrint);
18
19     add(panel);
20
21     // Create and register anonymous inner-class listener
22     jbtNew.addActionListener(
23       new ActionListener() {
24         public void actionPerformed(ActionEvent e) {
25           System.out.println("Process New");
26         }
27       }
28     );
```

anonymous listener
handle event

```
29
30      jbtOpen.addActionListener(
31        new ActionListener() {
32          public void actionPerformed(ActionEvent e) {
33            System.out.println("Process Open");
34          }
35        }
36      );
37
38      jbtSave.addActionListener(
39        new ActionListener() {
40          public void actionPerformed(ActionEvent e) {
41            System.out.println("Process Save");
42          }
43        }
44      );
45
46      jbtPrint.addActionListener(
47        new ActionListener() {
48          public void actionPerformed(ActionEvent e) {
49            System.out.println("Process Print");
50          }
51        }
52      );
53    }
54
55    /** Main method */
56    public static void main(String[] args) {
57      JFrame frame = new AnonymousListenerDemo();
58      frame.setTitle("AnonymousListenerDemo");
59      frame.setLocationRelativeTo(null); // Center the frame
60      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
61      frame.pack();
62      frame.setVisible(true);
63    }
64  }
```

The program creates four listeners using anonymous inner classes (lines 22–52). Without using anonymous inner classes, you would have to create four separate classes. An anonymous listener works the same way as an inner class listener. The program is condensed using an anonymous inner class.

Anonymous inner classes are compiled into `OuterClassName$#.class`, where `#` starts at `1` and is incremented for each anonymous class the compiler encounters. In this example, the anonymous inner class is compiled into `AnonymousListenerDemo$1.class`, `AnonymousListenerDemo$2.class`, `AnonymousListenerDemo$3.class`,and `AnonymousListenerDemo$4.class`.

pack()

Instead of using the `setSize` method to set the size for the frame, the program uses the `pack()` method (line 61), which automatically sizes the frame according to the size of the components placed in it.

16.6 Alternative Ways of Defining Listener Classes

There are many other ways to define the listener classes. For example, you may rewrite Listing 16.3 by creating just one listener, register the listener with the buttons, and let the listener detect the event source—i.e., which button fires the event—as shown in Listing 16.4.

LISTING 16.4 DetectSourceDemo.java

```
1 import javax.swing.*;
2 import java.awt.event.*;
3
4 public class DetectSourceDemo extends JFrame {
5   // Create four buttons
6   private JButton jbtNew = new JButton("New");
7   private JButton jbtOpen = new JButton("Open");
8   private JButton jbtSave = new JButton("Save");
9   private JButton jbtPrint = new JButton("Print");
10
11  public DetectSourceDemo() {
12    // Create a panel to hold buttons
13    JPanel panel = new JPanel();
14    panel.add(jbtNew);
15    panel.add(jbtOpen);
16    panel.add(jbtSave);
17    panel.add(jbtPrint);
18
19    add(panel);
20
21    // Create a listener
22    ButtonListener listener = new ButtonListener();                    create listener
23
24    // Register listener with buttons
25    jbtNew.addActionListener(listener);                                register listener
26    jbtOpen.addActionListener(listener);
27    jbtSave.addActionListener(listener);
28    jbtPrint.addActionListener(listener);
29  }
30
31  class ButtonListener implements ActionListener {                     listener class
32    public void actionPerformed(ActionEvent e) {                       handle event
33      if (e.getSource() == jbtNew)
34        System.out.println("Process New");
35      else if (e.getSource() == jbtOpen)
36        System.out.println("Process Open");
37      else if (e.getSource() == jbtSave)
38        System.out.println("Process Save");
39      else if (e.getSource() == jbtPrint)
40        System.out.println("Process Print");
41    }
42  }
43
44  /** Main method */
45  public static void main(String[] args) {
46    JFrame frame = new DetectSourceDemo();
47    frame.setTitle("DetectSourceDemo");
48    frame.setLocationRelativeTo(null); // Center the frame
49    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
50    frame.pack();
51    frame.setVisible(true);
52  }
53 }
```

This program defines just one inner listener class (lines 31–42), creates a listener from the class (line 22), and registers it to four buttons (lines 25–28). When a button is clicked, the button fires an `ActionEvent` and invokes the listener's `actionPerformed` method. The `actionPerformed` method checks the source of the event using the `getSource()` method for the event (lines 33, 35, 37, 39) and determines which button fired the event.

You may also rewrite Listing 16.3 by defining the custom frame class that implements `ActionListener`, as shown in Listing 16.5.

LISTING 16.5 FrameAsListenerDemo.java

implement `ActionListener`

register listeners

handle event

```
1  import javax.swing.*;
2  import java.awt.event.*;
3
4  public class FrameAsListenerDemo extends JFrame
5      implements ActionListener {
6    // Create four buttons
7    private JButton jbtNew = new JButton("New");
8    private JButton jbtOpen = new JButton("Open");
9    private JButton jbtSave = new JButton("Save");
10   private JButton jbtPrint = new JButton("Print");
11
12   public FrameAsListenerDemo() {
13     // Create a panel to hold buttons
14     JPanel panel = new JPanel();
15     panel.add(jbtNew);
16     panel.add(jbtOpen);
17     panel.add(jbtSave);
18     panel.add(jbtPrint);
19
20     add(panel);
21
22     // Register listener with buttons
23     jbtNew.addActionListener(this);
24     jbtOpen.addActionListener(this);
25     jbtSave.addActionListener(this);
26     jbtPrint.addActionListener(this);
27   }
28
29   /** Implement actionPerformed */
30   public void actionPerformed(ActionEvent e) {
31     if (e.getSource() == jbtNew)
32       System.out.println("Process New");
33     else if (e.getSource() == jbtOpen)
34       System.out.println("Process Open");
35     else if (e.getSource() == jbtSave)
36       System.out.println("Process Save");
37     else if (e.getSource() == jbtPrint)
38       System.out.println("Process Print");
39   }
40
41   /** Main method */
42   public static void main(String[] args) {
43     JFrame frame = new FrameAsListenerDemo();
44     frame.setTitle("FrameAsListenerDemo");
45     frame.setLocationRelativeTo(null); // Center the frame
46     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
47     frame.pack();
48     frame.setVisible(true);
49   }
50 }
```

The frame class extends `JFrame` and implements `ActionListener` (line 5). So the class is a listener class for action events. The listener is registered to four buttons (lines 23–26). When a button is clicked, the button fires an `ActionEvent` and invokes the listener's `actionPerformed` method. The `actionPerformed` method checks the source of the event using the `getSource()` method for the event (lines 31, 33, 35, 37) and determines which button fired the event.

This design is not desirable because it places too many responsibilities into one class. It is better to design a listener class that is solely responsible for handling events. This design makes the code easy to read and easy to maintain.

You can define listener classes in many ways. Which way is preferred? Defining listener classes using inner class or anonymous inner class has become a standard for event-handling programming because it generally provides clear, clean, and concise code. So, we will consistently use it in this book.

16.7 Problem: Loan Calculator

Now you can write the program for the loan-calculator problem presented in the introduction of this chapter. Here are the major steps in the program:

1. Create the user interface, as shown in Figure 16.9.

   a. Create a panel of a `GridLayout` with `5` rows and `2` columns. Add labels and text fields into the panel. Set a title "Enter loan amount, interest rate, and years" for the panel.

   b. Create another panel with a `FlowLayout(FlowLayout.RIGHT)` and add a button into the panel.

   c. Add the first panel to the center of the frame and the second panel to the south side of the frame.

2. Process the event.

   Create and register the listener for processing the button-clicking action event. The handler obtains the user input on loan, interest rate, and number of years, computes the monthly and total payments, and displays the values in the text fields.

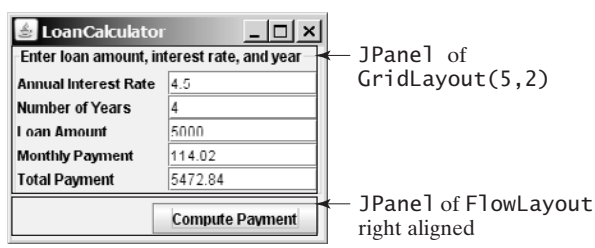


FIGURE 16.9 The program computes loan payments.

The complete program is given in Listing 16.6.

LISTING 16.6 LoanCalculator.java

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4 import javax.swing.border.TitledBorder;
5
6 public class LoanCalculator extends JFrame {
7   // Create text fields for interest rate,
```

text fields

```
  // year, loan amount, monthly payment, and total payment
  private JTextField jtfAnnualInterestRate = new JTextField();
  private JTextField jtfNumberOfYears = new JTextField();
  private JTextField jtfLoanAmount = new JTextField();
  private JTextField jtfMonthlyPayment = new JTextField();
  private JTextField jtfTotalPayment = new JTextField();
```

button

```
  // Create a Compute Payment button
  private JButton jbtComputeLoan = new JButton("Compute Payment");
```

create UI

```
  public LoanCalculator() {
    // Panel p1 to hold labels and text fields
    JPanel p1 = new JPanel(new GridLayout(5, 2));
    p1.add(new JLabel("Annual Interest Rate"));
    p1.add(jtfAnnualInterestRate);
    p1.add(new JLabel("Number of Years"));
    p1.add(jtfNumberOfYears);
    p1.add(new JLabel("Loan Amount"));
    p1.add(jtfLoanAmount);
    p1.add(new JLabel("Monthly Payment"));
    p1.add(jtfMonthlyPayment);
    p1.add(new JLabel("Total Payment"));
    p1.add(jtfTotalPayment);
    p1.setBorder(new
      TitledBorder("Enter loan amount, interest rate, and year"));
```

add to frame

```
    // Panel p2 to hold the button
    JPanel p2 = new JPanel(new FlowLayout(FlowLayout.RIGHT));
    p2.add(jbtComputeLoan);

    // Add the panels to the frame
    add(p1, BorderLayout.CENTER);
    add(p2, BorderLayout.SOUTH);
```

register listener

```
    // Register listener
    jbtComputeLoan.addActionListener(new ButtonListener());
  }

  /** Handle the Compute Payment button */
  private class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      // Get values from text fields
      double interest =
```

get input

```
        Double.parseDouble(jtfAnnualInterestRate.getText());
      int year =
        Integer.parseInt(jtfNumberOfYears.getText());
      double loanAmount =
        Double.parseDouble(jtfLoanAmount.getText());
```

create loan

```
      // Create a loan object
      Loan loan = new Loan(interest, year, loanAmount);
```

set result

```
      // Display monthly payment and total payment
      jtfMonthlyPayment.setText(String.format("%.2f",
        loan.getMonthlyPayment()));
      jtfTotalPayment.setText(String.format("%.2f",
        loan.getTotalPayment()));
    }
  }
```

```
68   public static void main(String[] args) {
69     LoanCalculator frame = new LoanCalculator();
70     frame.pack();
71     frame.setTitle("LoanCalculator");
72     frame.setLocationRelativeTo(null); // Center the frame
73     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
74     frame.setVisible(true);
75   }
76 }
```

The user interface is created in the constructor (lines 18–44). The button is the source of the event. A listener is created and registered with the button (line 43).

The listener class (lines 47–66) implements the `actionPerformed` method. When the button is clicked, the `actionPerformed` method is invoked to get the interest rate (line 51), number of years (line 53), and loan amount (line 55). Invoking `jtfAnnualInterestRate.getText()` returns the string text in the `jtfAnnualInterestRate` text field. The loan is used for computing the loan payments. This class was introduced in Listing 10.2, Loan.java. Invoking `loan.getMonthlyPayment()` returns the monthly payment for the loan. The `String.format` method uses the `printf` like syntax to format a number into a desirable format. Invoking the `setText` method on a text field sets a string value in the text field (line 61).

16.8 Window Events

The preceding sections used action events. Other events can be processed similarly. This section gives an example of handling `WindowEvent`. Any subclass of the `Window` class can fire the following window events: window opened, closing, closed, activated, deactivated, iconified, and deiconified. The program in Listing 16.7 creates a frame, listens to the window events, and displays a message to indicate the occurring event. Figure 16.10 shows a sample run of the program.



FIGURE 16.10 The window events are displayed on the console when you run the program from the command prompt.

LISTING 16.7 TestWindowEvent.java

```
 1 import java.awt.event.*;
 2 import javax.swing.JFrame;
 3
 4 public class TestWindowEvent extends JFrame {
 5   public static void main(String[] args) {
 6     TestWindowEvent frame = new TestWindowEvent();
 7     frame.setSize(220, 80);
 8     frame.setLocationRelativeTo(null); // Center the frame
 9     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10     frame.setTitle("TestWindowEvent");
11     frame.setVisible(true);
12   }
13
14   public TestWindowEvent() {
15     addWindowListener(new WindowListener() {
16       /**
17        * Handler for window-deiconified event
```

```
18          * Invoked when a window is changed from a minimized
19          * to a normal state.
20          */
21         public void windowDeiconified(WindowEvent event) {
22           System.out.println("Window deiconified");
23         }
24
25         /**
26          * Handler for window-iconified event
27          * Invoked when a window is changed from a normal to a
28          * minimized state. For many platforms, a minimized window
29          * is displayed as the icon specified in the window's
30          * iconImage property.
31          */
32         public void windowIconified(WindowEvent event) {
33           System.out.println("Window iconified");
34         }
35
36         /**
37          * Handler for window-activated event
38          * Invoked when the window is set to be the user's
39          * active window, which means the window (or one of its
40          * subcomponents) will receive keyboard events.
41          */
42         public void windowActivated(WindowEvent event) {
43           System.out.println("Window activated");
44         }
45
46         /**
47          * Handler for window-deactivated event
48          * Invoked when a window is no longer the user's active
49          * window, which means that keyboard events will no longer
50          * be delivered to the window or its subcomponents.
51          */
52         public void windowDeactivated(WindowEvent event) {
53           System.out.println("Window deactivated");
54         }
55
56         /**
57          * Handler for window-opened event
58          * Invoked the first time a window is made visible.
59          */
60         public void windowOpened(WindowEvent event) {
61           System.out.println("Window opened");
62         }
63
64         /**
65          * Handler for window-closing event
66          * Invoked when the user attempts to close the window
67          * from the window's system menu.  If the program does not
68          * explicitly hide or dispose the window while processing
69          * this event, the window-closing operation will be cancelled.
70          */
71         public void windowClosing(WindowEvent event) {
72           System.out.println("Window closing");
73         }
74
75         /**
76          * Handler for window-closed event
77          * Invoked when a window has been closed as the result
```

implement handler

implement handler

implement handler

implement handler

implement handler

```
78         * of calling dispose on the window.
79         */
80        public void windowClosed(WindowEvent event) {
81          System.out.println("Window closed");
82        }
83      });
84    }
85 }
```

implement handler

The `WindowEvent` can be fired by the `Window` class or by any subclass of `Window`. Since `JFrame` is a subclass of `Window`, it can fire `WindowEvent`.

`TestWindowEvent` extends `JFrame` and implements `WindowListener`. The `WindowListener` interface defines several abstract methods (`windowActivated`, `windowClosed`, `windowClosing`, `windowDeactivated`, `windowDeiconified`, `windowIconified`, `windowOpened`) for handling window events when the window is activated, closed, closing, deactivated, deiconified, iconified, or opened.

When a window event, such as activation, occurs, the `windowActivated` method is invoked. Implement the `windowActivated` method with a concrete response if you want the event to be processed.

16.9 Listener Interface Adapters

Because the methods in the `WindowListener` interface are abstract, you must implement all of them even if your program does not care about some of the events. For convenience, Java provides support classes, called *convenience adapters*, which provide default implementations for all the methods in the listener interface. The default implementation is simply an empty body. Java provides convenience listener adapters for every AWT listener interface with multiple handlers. A convenience listener adapter is named *X*Adapter for *X*Listener. For example, `WindowAdapter` is a convenience listener adapter for `WindowListener`. Table 16.3 lists the convenience adapters.

convenience adapter

TABLE 16.3 Convenience Adapters

Adapter	*Interface*
`WindowAdapter`	`WindowListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`KeyAdapter`	`KeyListener`
`ContainerAdapter`	`ContainerListener`
`ComponentAdapter`	`ComponentListener`
`FocusAdapter`	`FocusListener`

Using `WindowAdapter`, the preceding example can be simplified as shown in Listing 16.8, if you are interested only in the window-activated event. The `WindowAdapter` class is used to create an anonymous listener instead of `WindowListener` (line 15). The `windowActivated` handler is implemented in line 16.

LISTING 16.8 AdapterDemo.java

```
1 import java.awt.event.*;
2 import javax.swing.JFrame;
3
4 public class AdapterDemo extends JFrame {
```

```
 5   public static void main(String[] args) {
 6     AdapterDemo frame = new AdapterDemo();
 7     frame.setSize(220, 80);
 8     frame.setLocationRelativeTo(null); // Center the frame
 9     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10     frame.setTitle("AdapterDemo");
11     frame.setVisible(true);
12   }
13
14   public AdapterDemo() {
15     addWindowListener(new WindowAdapter() {
16       public void windowActivated(WindowEvent event) {
17         System.out.println("Window activated");
18       }
19     });
20   }
21 }
```

register listener
implement handler

16.10 Mouse Events

A mouse event is fired whenever a mouse is pressed, released, clicked, moved, or dragged on a component. The `MouseEvent` object captures the event, such as the number of clicks associated with it or the location (`x`- and `y`-coordinates) of the mouse, as shown in Figure 16.11.

Since the `MouseEvent` class inherits `InputEvent`, you can use the methods defined in the `InputEvent` class on a `MouseEvent` object.

`Point` class

The `java.awt.Point` class represents a point on a component. The class contains two public variables, `x` and `y`, for coordinates. To create a `Point`, use the following constructor:

```
Point(int x, int y)
```

This constructs a `Point` object with the specified `x`- and `y`-coordinates. Normally, the data fields in a class should be private, but this class has two public data fields.

Java provides two listener interfaces, `MouseListener` and `MouseMotionListener`, to handle mouse events, as shown in Figure 16.12. Implement the `MouseListener` interface to

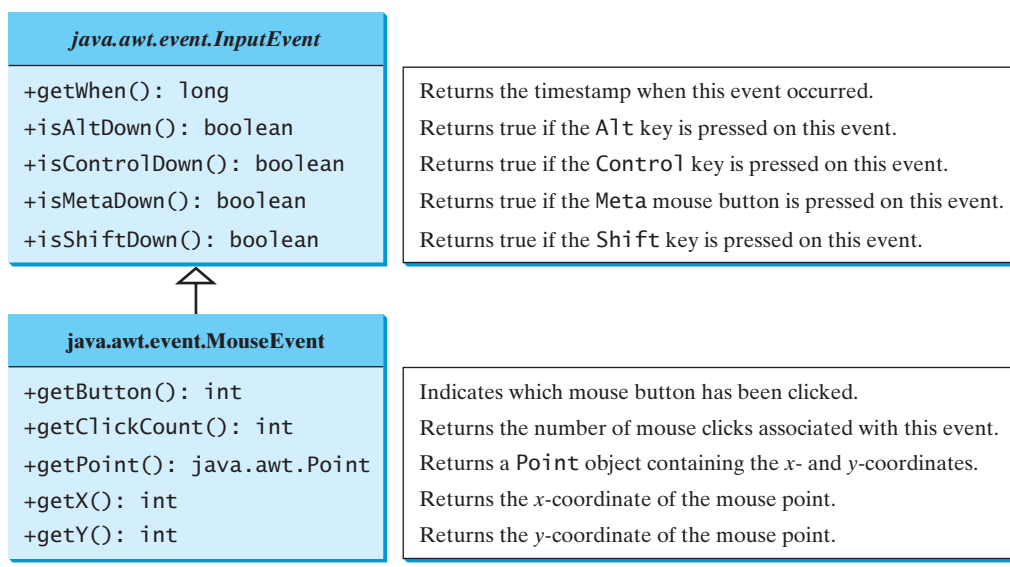

java.awt.event.InputEvent
+getWhen(): long
+isAltDown(): boolean
+isControlDown(): boolean
+isMetaDown(): boolean
+isShiftDown(): boolean
Returns the timestamp when this event occurred.
Returns true if the Alt key is pressed on this event.
Returns true if the Control key is pressed on this event.
Returns true if the Meta mouse button is pressed on this event.
Returns true if the Shift key is pressed on this event.
java.awt.event.MouseEvent
+getButton(): int
+getClickCount(): int
+getPoint(): java.awt.Point
+getX(): int
+getY(): int
Indicates which mouse button has been clicked.
Returns the number of mouse clicks associated with this event.
Returns a Point object containing the x- and y-coordinates.
Returns the x-coordinate of the mouse point.
Returns the y-coordinate of the mouse point.

FIGURE 16.11 The `MouseEvent` class encapsulates information for mouse events.

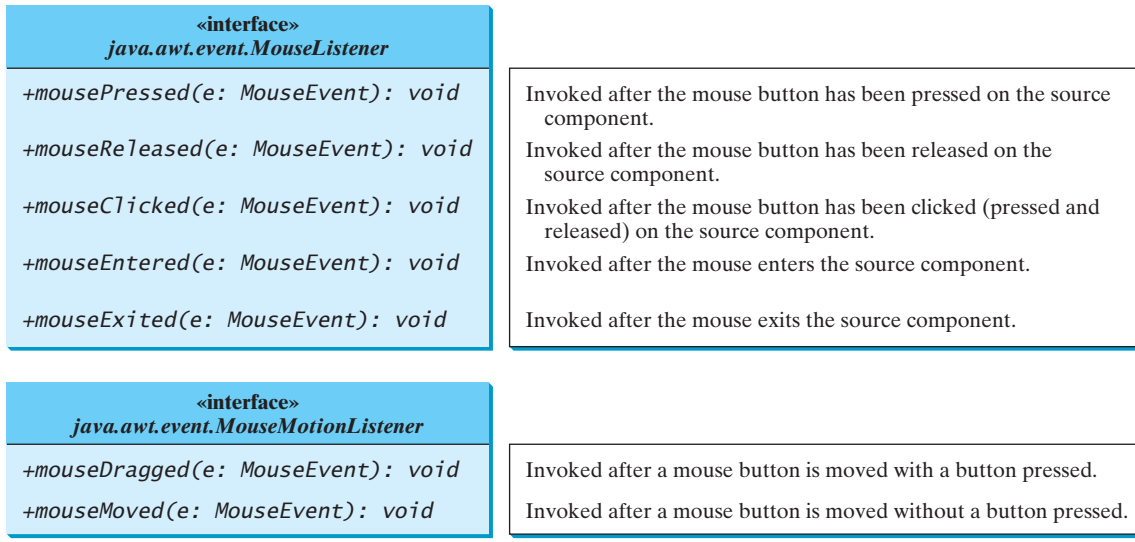

«interface»
java.awt.event.MouseListener
+mousePressed(e: MouseEvent): void
+mouseReleased(e: MouseEvent): void
+mouseClicked(e: MouseEvent): void
+mouseEntered(e: MouseEvent): void
+mouseExited(e: MouseEvent): void
Invoked after the mouse button has been pressed on the source component.
Invoked after the mouse button has been released on the source component.
Invoked after the mouse button has been clicked (pressed and released) on the source component.
Invoked after the mouse enters the source component.
Invoked after the mouse exits the source component.
«interface»
java.awt.event.MouseMotionListener
+mouseDragged(e: MouseEvent): void
+mouseMoved(e: MouseEvent): void
Invoked after a mouse button is moved with a button pressed.
Invoked after a mouse button is moved without a button pressed.

FIGURE 16.12 The `MouseListener` interface handles mouse pressed, released, clicked, entered, and exited events. The `MouseMotionListener` interface handles mouse dragged and moved events.

listen for such actions as pressing, releasing, entering, exiting, or clicking the mouse, and implement the `MouseMotionListener` interface to listen for such actions as dragging or moving the mouse.

16.10.1 Example: Moving a Message on a Panel Using a Mouse

This example writes a program that displays a message in a panel, as shown in Listing 16.9. You can use the mouse to move the message. The message moves as the mouse drags and is always displayed at the mouse point. A sample run of the program is shown in Figure 16.13.



FIGURE 16.13 You can move the message by dragging the mouse.

LISTING 16.9 MoveMessageDemo.java



Video Note
Move message using the mouse

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class MoveMessageDemo extends JFrame {
6    public MoveMessageDemo() {
7      // Create a MovableMessagePanel instance for moving a message
8      MovableMessagePanel p = new MovableMessagePanel
9        ("Welcome to Java");
10
11     // Place the message panel in the frame
12     setLayout(new BorderLayout());
13     add(p);
14   }
15
```

create a panel

```
16   /** Main method */
17   public static void main(String[] args) {
18     MoveMessageDemo frame = new MoveMessageDemo();
19     frame.setTitle("MoveMessageDemo");
20     frame.setSize(200, 100);
21     frame.setLocationRelativeTo(null); // Center the frame
22     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23     frame.setVisible(true);
24   }
25
26   // Inner class: MovableMessagePanel draws a message
27   static class MovableMessagePanel extends JPanel {
28     private String message = "Welcome to Java";
29     private int x = 20;
30     private int y = 20;
31
32     /** Construct a panel to draw string s */
33     public MovableMessagePanel(String s) {
34       message = s;
35       addMouseMotionListener(new MouseMotionAdapter() {
36         /** Handle mouse-dragged event */
37         public void mouseDragged(MouseEvent e) {
38           // Get the new location and repaint the screen
39           x = e.getX();
40           y = e.getY();
41           repaint();
42         }
43       });
44     }
45
46     /** Paint the component */
47     protected void paintComponent(Graphics g) {
48       super.paintComponent(g);
49       g.drawString(message, x, y);
50     }
51   }
52 }
```

inner class

set a new message

anonymous listener

override handler

new location

repaint

paint message

The `MovableMessagePanel` class extends `JPanel` to draw a message (line 27). Additionally, it handles redisplaying the message when the mouse is dragged. This class is defined as an inner class inside the main class because it is used only in this class. Furthermore, the inner class is defined static because it does not reference any instance members of the main class.

The `MouseMotionListener` interface contains two handlers, `mouseMoved` and `mouseDragged`, for handling mouse-motion events. When you move the mouse with the button pressed, the `mouseDragged` method is invoked to repaint the viewing area and display the message at the mouse point. When you move the mouse without pressing the button, the `mouseMoved` method is invoked.

Because the listener is interested only in the mouse-dragged event, the anonymous inner-class listener extends `MouseMotionAdapter` to override the `mouseDragged` method. If the inner class implemented the `MouseMotionListener` interface, you would have to implement all of the handlers, even if your listener did not care about some of the events.

The `mouseDragged` method is invoked when you move the mouse with a button pressed. This method obtains the mouse location using `getX` and `getY` methods (lines 39–40) in the `MouseEvent` class. This becomes the new location for the message. Invoking the `repaint()` method (line 41) causes `paintComponent` to be invoked (line 47), which displays the message in a new location.

16.11 Key Events

Key events enable the use of the keys to control and perform actions or get input from the keyboard. A key event is fired whenever a key is pressed, released, or typed on a component. The `KeyEvent` object describes the nature of the event (namely, that a key has been pressed, released, or typed) and the value of the key, as shown in Figure 16.14. Java provides the `KeyListener` interface to handle key events, as shown in Figure 16.15.

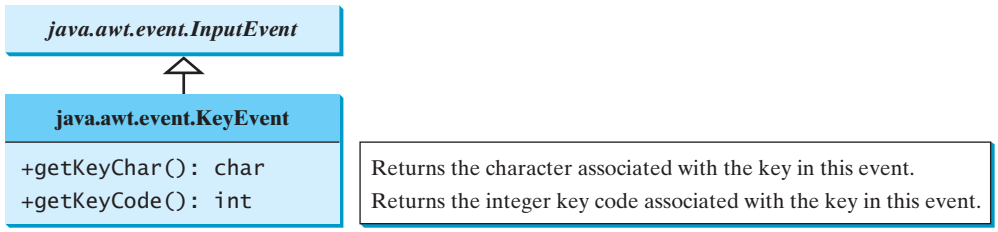

java.awt.event.InputEvent
java.awt.event.KeyEvent
+getKeyChar(): char
+getKeyCode(): int
Returns the character associated with the key in this event.
Returns the integer key code associated with the key in this event.

FIGURE 16.14 The `KeyEvent` class encapsulates information about key events.

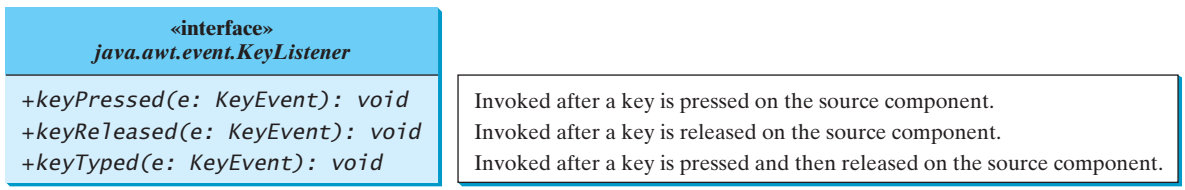

«interface»
java.awt.event.KeyListener
+keyPressed(e: KeyEvent): void
+keyReleased(e: KeyEvent): void
+keyTyped(e: KeyEvent): void
Invoked after a key is pressed on the source component.
Invoked after a key is released on the source component.
Invoked after a key is pressed and then released on the source component.

FIGURE 16.15 The `KeyListener` interface handles key pressed, released, and typed events.

The `keyPressed` handler is invoked when a key is pressed, the `keyReleased` handler is invoked when a key is released, and the `keyTyped` handler is invoked when a Unicode character is entered. If a key does not have a Unicode (e.g., function keys, modifier keys, action keys, and control keys), the `keyTyped` handler will be not be invoked.

Every key event has an associated key character or key code that is returned by the `getKeyChar()` or `getKeyCode()` method in `KeyEvent`. The key codes are constants defined in Table 16.4. For a key of the Unicode character, the key code is the same as the Unicode value.

TABLE 16.4 Key Constants

Constant	*Description*	*Constant*	*Description*
`VK_HOME`	The Home key	`VK_SHIFT`	The Shift key
`VK_END`	The End key	`VK_BACK_SPACE`	The Backspace key
`VK_PGUP`	The Page Up key	`VK_CAPS_LOCK`	The Caps Lock key
`VK_PGDN`	The Page Down key	`VK_NUM_LOCK`	The Num Lock key
`VK_UP`	The up-arrow key	`VK_ENTER`	The Enter key
`VK_DOWN`	The down-arrow key	`VK_UNDEFINED`	The keyCode unknown
`VK_LEFT`	The left-arrow key	`VK_F1` to `VK_F12`	The function keys from F1 to F12
`VK_RIGHT`	The right-arrow key		
`VK_ESCAPE`	The Esc key	`VK_0` to `VK_9`	The number keys from 0 to 9
`VK_TAB`	The Tab key	`VK_A` to `VK_Z`	The letter keys from A to Z
`VK_CONTROL`	The Control key		

For the key-pressed and key-released events, `getKeyCode()` returns the value as defined in the table. For the key-typed event, `getKeyCode()` returns `VK_UNDEFINED`, while `getKeyChar()` returns the character entered.

The program in Listing 16.10 displays a user-input character. The user can move the character up, down, left, and right, using the arrow keys `VK_UP`, `VK_DOWN`, `VK_LEFT`, and `VK_RIGHT`. Figure 16.16 contains a sample run of the program.



FIGURE 16.16 The program responds to key events by displaying a character and moving it up, down, left, or right.

LISTING 16.10 KeyEventDemo.java

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class KeyEventDemo extends JFrame {
6    private KeyboardPanel keyboardPanel = new KeyboardPanel();
7
8    /** Initialize UI */
9    public KeyEventDemo() {
10     // Add the keyboard panel to accept and display user input
11     add(keyboardPanel);
12
13     // Set focus
14     keyboardPanel.setFocusable(true);
15   }
16
17   /** Main method */
18   public static void main(String[] args) {
19     KeyEventDemo frame = new KeyEventDemo();
20     frame.setTitle("KeyEventDemo");
21     frame.setSize(300, 300);
22     frame.setLocationRelativeTo(null); // Center the frame
23     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24     frame.setVisible(true);
25   }
26
27   // Inner class: KeyboardPanel for receiving key input
28   static class KeyboardPanel extends JPanel {
29     private int x = 100;
30     private int y = 100;
31     private char keyChar = 'A'; // Default key
32
33     public KeyboardPanel() {
34       addKeyListener(new KeyAdapter() {
35         public void keyPressed(KeyEvent e) {
36           switch (e.getKeyCode()) {
37             case KeyEvent.VK_DOWN: y += 10; break;
38             case KeyEvent.VK_UP: y -= 10; break;
39             case KeyEvent.VK_LEFT: x -= 10; break;
40             case KeyEvent.VK_RIGHT: x += 10; break;
41             default: keyChar = e.getKeyChar();
42           }
```

create a panel

focusable

inner class

register listener
override handler

get the key pressed

```
43
44            repaint();                                                    repaint
45          }
46        });
47      }
48
49      /** Draw the character */
50      protected void paintComponent(Graphics g) {
51        super.paintComponent(g);
52
53        g.setFont(new Font("TimesRoman", Font.PLAIN, 24));
54        g.drawString(String.valueOf(keyChar), x, y);                      redraw character
55      }
56    }
57 }
```

The `KeyboardPanel` class extends `JPanel` to display a character (line 28). This class is defined as an inner class inside the main class, because it is used only in this class. Furthermore, the inner class is defined static, because it does not reference any instance members of the main class.

Because the program gets input from the keyboard, it listens for `KeyEvent` and extends `KeyAdapter` to handle key input (line 34).

When a key is pressed, the `keyPressed` handler is invoked. The program uses `e.getKeyCode()` to obtain the key code and `e.getKeyChar()` to get the character for the key. When a nonarrow key is pressed, the character is displayed (line 41). When an arrow key is pressed, the character moves in the direction indicated by the arrow key (lines 37–40).

Only a focused component can receive `KeyEvent`. To make a component focusable, set its `isFocusable` property to `true` (line 14). focusable

Every time the component is repainted, a new font is created for the `Graphics` object in line 53. This is not efficient. It is better to create the font once as a data field. efficient?

16.12 Animation Using the `Timer` Class

Not all source objects are GUI components. The `javax.swing.Timer` class is a source component that fires an `ActionEvent` at a predefined rate. Figure 16.17 lists some of the methods in the class.

javax.swing.Timer	
+Timer(delay: int, listener: ActionListener)	Creates a Timer object with a specified delay in milliseconds and an ActionListener.
+addActionListener(listener: ActionListener): void	Adds an ActionListener to the timer.
+start(): void	Starts this timer.
+stop(): void	Stops this timer.
+setDelay(delay: int): void	Sets a new delay value for this timer.

FIGURE 16.17 A `Timer` object fires an `ActionEvent` at a fixed rate.

A `Timer` object serves as the source of an `ActionEvent`. The listeners must be instances of `ActionListener` and registered with a `Timer` object. You create a `Timer` object using its sole constructor with a delay and a listener, where `delay` specifies the number of milliseconds between two action events. You can add additional listeners using the `addActionListener`

method and adjust the `delay` using the `setDelay` method. To start the timer, invoke the `start()` method. To stop the timer, invoke the `stop()` method.

The `Timer` class can be used to control animations. For example, you can use it to display a moving message, as shown in Figure 16.18, with the code in Listing 16.11.



FIGURE 16.18 A message moves in the panel.

LISTING 16.11 AnimationDemo.java

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class AnimationDemo extends JFrame {
6    public AnimationDemo() {
7      // Create a MovingMessagePanel for displaying a moving message
8      add(new MovingMessagePanel("message moving?"));
9    }
10
11   /** Main method */
12   public static void main(String[] args) {
13     AnimationDemo frame = new AnimationDemo();
14     frame.setTitle("AnimationDemo");
15     frame.setSize(280, 100);
16     frame.setLocationRelativeTo(null); // Center the frame
17     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18     frame.setVisible(true);
19   }
20
21   // Inner class: Displaying a moving message
22   static class MovingMessagePanel extends JPanel {
23     private String message = "Welcome to Java";
24     private int xCoordinate = 0;
25     private int yCoordinate = 20;
26
27     public MovingMessagePanel(String message) {
28       this.message = message;
29
30       // Create a timer
31       Timer timer = new Timer(1000, new TimerListener());
32       timer.start();
33     }
34
35     /** Paint message */
36     protected void paintComponent(Graphics g) {
37       super.paintComponent(g);
38
39       if (xCoordinate > getWidth()) {
40         xCoordinate = -20;
41       }
42       xCoordinate += 5;
43       g.drawString(message, xCoordinate, yCoordinate);
44     }
```

create panel

set message

create timer
start timer

reset x-coordinate

move message

```
45
46      class TimerListener implements ActionListener {
47        /** Handle ActionEvent */
48        public void actionPerformed(ActionEvent e) {
49          repaint();
50        }
51      }
52    }
53  }
```

listener class
event handler
repaint

The `MovingMessagePanel` class extends `JPanel` to display a message (line 22). This class is defined as an inner class inside the main class, because it is used only in this class. Furthermore, the inner class is defined static, because it does not reference any instance members of the main class.

An inner class listener is defined in line 46 to listen for `ActionEvent`. Line 31 creates a `Timer` for the listener. The timer is started in line 32. The timer fires an `ActionEvent` every second, and the listener responds in line 49 to repaint the panel. When a panel is painted, its `x`-coordinate is increased (line 42), so the message is displayed to the right. When the `x`-coordinate exceeds the bound of the panel, it is reset to `-20` (line 40), so the message continues moving from left to right.

In §15.12, "Case Study: The `StillClock` Class," you drew a `StillClock` to show the current time. The clock does not tick after it is displayed. What can you do to make the clock display a new current time every second? The key to making the clock tick is to repaint it every second with a new current time. You can use a timer to control the repainting of the clock with the code in Listing 16.12.

LISTING 16.12 ClockAnimation.java



Video Note
Animate a clock

```
1  import java.awt.event.*;
2  import javax.swing.*;
3
4  public class ClockAnimation extends JFrame {
5    private StillClock clock = new StillClock();
6
7    public ClockAnimation() {
8      add(clock);
9
10     // Create a timer with delay 1000 ms
11     Timer timer = new Timer(1000, new TimerListener());
12     timer.start();
13   }
14
15   private class TimerListener implements ActionListener {
16     /** Handle the action event */
17     public void actionPerformed(ActionEvent e) {
18       // Set new time and repaint the clock to display current time
19       clock.setCurrentTime();
20       clock.repaint();
21     }
22   }
23
24   /** Main method */
25   public static void main(String[] args) {
26     JFrame frame = new ClockAnimation();
27     frame.setTitle("ClockAnimation");
28     frame.setSize(200, 200);
29     frame.setLocationRelativeTo(null); // Center the frame
30     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

create a clock
create a timer
start timer
listener class
implement handler
set new time
repaint

```
31      frame.setVisible(true);
32    }
33 }
```

The program displays a running clock, as shown in Figure 16.19. `ClockAnimation` creates a `StillClock` (line 5). Line 11 creates a `Timer` for a `ClockAnimation`. The timer is started in line 12. The timer fires an `ActionEvent` every second, and the listener responds to set a new time (line 19) and repaint the clock (line 20). The `setCurrentTime()` method defined in `StillClock` sets the current time in the clock.

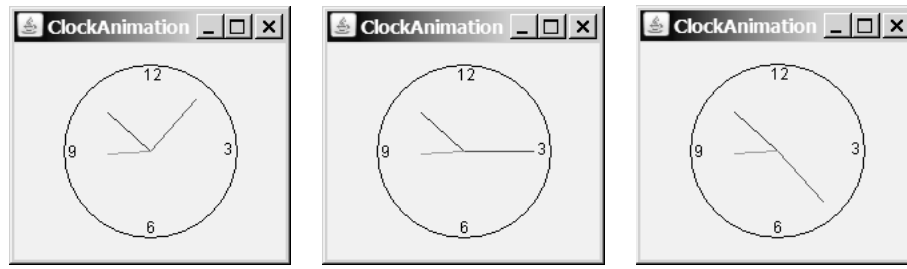


FIGURE 16.19 A live clock is displayed in the panel.

KEY TERMS

anonymous inner class 542
convenience listener adapter 551
event 534
event delegation 535
event handler 559
event listener 535
event-listener interface 536
event object 535
event registration 535
event source (source object) 535
event-driven programming 534
inner class 554

CHAPTER SUMMARY

1. The root class of the event classes is `java.util.EventObject`. The subclasses of `EventObject` deal with special types of events, such as action events, window events, component events, mouse events, and key events. You can identify the source object of an event using the `getSource()` instance method in the `EventObject` class. If a component can fire an event, any subclass of the component can fire the same type of event.

2. The listener object's class must implement the corresponding event-listener interface. Java provides a listener interface for every event class. The listener interface is usually named `XListener` for `XEvent`, with the exception of `MouseMotionListener`. For example, the corresponding listener interface for `ActionEvent` is `ActionListener`; each listener for `ActionEvent` should implement the `ActionListener` interface. The listener interface contains the method(s), known as the *handler(s)*, which process the events.

3. The listener object must be registered by the source object. Registration methods depend on the event type. For `ActionEvent`, the method is `addActionListener`. In general, the method is named `addXListener` for `XEvent`.

4. An *inner class*, or *nested class,* is defined within the scope of another class. An inner class can reference the data and methods defined in the outer class in which it nests, so you need not pass the reference of the outer class to the constructor of the inner class.

5. Convenience adapters are support classes that provide default implementations for all the methods in the listener interface. Java provides convenience listener adapters for every AWT listener interface with multiple handlers. A convenience listener adapter is named *X*Adapter for *X*Listener.

6. A source object may fire several types of events. For each event, the source object maintains a list of registered listeners and notifies them by invoking the *handler* on the listener object to process the event.

7. A mouse event is fired whenever a mouse is clicked, released, moved, or dragged on a component. The mouse-event object captures the event, such as the number of clicks associated with it or the location (`x-` and `y`-coordinates) of the mouse point.

8. Java provides two listener interfaces, `MouseListener` and `MouseMotionListener`, to handle mouse events, implement the `MouseListener` interface to listen for such actions as mouse pressed, released, clicked, entered, or exited, and implement the `MouseMotionListener` interface to listen for such actions as mouse dragged or moved.

9. A `KeyEvent` object describes the nature of the event (namely, that a key has been pressed, released, or typed) and the value of the key.

10. The `keyPressed` handler is invoked when a key is pressed, the `keyReleased` handler is invoked when a key is released, and the `keyTyped` handler is invoked when a Unicode character key is entered. If a key does not have a Unicode (e.g., function keys, modifier keys, action keys, and control keys), the `keyTyped` handler will be not be invoked.

11. You can use the `Timer` class to control Java animations. A timer fires an `ActionEvent` at a fixed rate. The listener updates the painting to simulate an animation.

Review Questions

Sections 16.2–16.3

16.1 Can a button fire a `WindowEvent`? Can a button fire a `MouseEvent`? Can a button fire an `ActionEvent`?

16.2 Why must a listener be an instance of an appropriate listener interface? Explain how to register a listener object and how to implement a listener interface.

16.3 Can a source have multiple listeners? Can a listener listen on multiple sources? Can a source be a listener for itself?

16.4 How do you implement a method defined in the listener interface? Do you need to implement all the methods defined in the listener interface?

Sections 16.4–16.9

16.5 Can an inner class be used in a class other than the class in which it nests?

16.6 Can the modifiers `public`, `private`, and `static` be used on inner classes?

16.7 If class `A` is an inner class in class `B`, what is the .class file for `A`? If class B contains two anonymous inner classes, what are the .class file names for these two classes?

16.8 What is wrong in the following code?

```
import java.swing.*;
import java.awt.*;

public class Test extends JFrame {
  public Test() {
    JButton jbtOK = new JButton("OK");
    add(jbtOK);
  }

  private class Listener
      implements ActionListener {
    public void actionPerform
        (ActionEvent e) {
      System.out.println
        (jbtOK.getActionCommand());
    }
  }

  /** Main method omitted */
}
```

(a)

```
import java.awt.event.*;
import javax.swing.*;

public class Test extends JFrame {
  public Test() {
    JButton jbtOK = new JButton("OK");
    add(jbtOK);
    jbtOK.addActionListener(
      new ActionListener() {
        public void actionPerformed
            (ActionEvent e) {
          System.out.println
            (jbtOK.getActionCommand());
        }
    }  // Something missing here
  }

  /** Main method omitted */
}
```

(b)

16.9 What is the difference between the `setSize(width, height)` method and the `pack()` method in `JFrame`?

Sections 16.10–16.11

16.10 What method do you use to get the source of an event? What method do you use to get the timestamp for an action event, a mouse event, or a key event? What method do you use to get the mouse-point position for a mouse event? What method do you use to get the key character for a key event?

16.11 What is the listener interface for mouse pressed, released, clicked, entered, and exited? What is the listener interface for mouse moved and dragged?

16.12 Does every key in the keyboard have a Unicode? Is a key code in the `KeyEvent` class equivalent to a Unicode?

16.13 Is the `keyPressed` handler invoked after a key is pressed? Is the `keyReleased` handler invoked after a key is released? Is the `keyTyped` handler invoked after *any* key is typed?

Section 16.12

16.14 How do you create a timer? How do you start a timer? How do you stop a timer?

16.15 Does the `Timer` class have a no-arg constructor? Can you add multiple listeners to a timer?

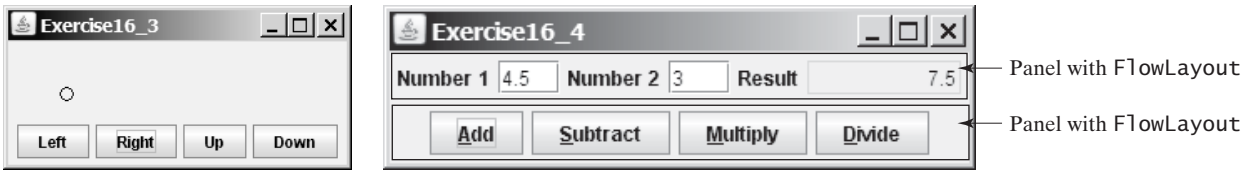
Programming Exercises

Sections 16.2–16.9

16.1 (*Finding which button has been clicked on the console*) Add the code to Exercise 12.1 that will display a message on the console indicating which button has been clicked.

16.2 (*Using `ComponentEvent`*) Any GUI component can fire a `ComponentEvent`. The `ComponentListener` defines the `componentMoved`, `componentResized`, `componentShown`, and `componentHidden` methods for processing component events. Write a test program to demonstrate `ComponentEvent`.

16.3* (*Moving the ball*) Write a program that moves the ball in a panel. You should define a panel class for displaying the ball and provide the methods for moving the button left, right, up, and down, as shown in Figure 16.20(a).



(a) (b)

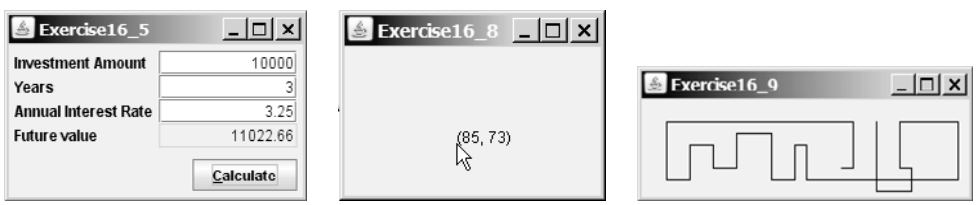
FIGURE 16.20 (a) Exercise 16.3 displays which button is clicked on a message panel. (b) The program performs addition, subtraction, multiplication, and division on double numbers.

16.4* (*Creating a simple calculator*) Write a program to perform add, subtract, multiply, and divide operations (see Figure 16.20(b)).

16.5* (*Creating an investment-value calculator*) Write a program that calculates the future value of an investment at a given interest rate for a specified number of years. The formula for the calculation is as follows:

$$\texttt{futureValue} = \texttt{investmentAmount} * (1 + \texttt{monthlyInterestRate})^{\texttt{years}*12}$$

Use text fields for interest rate, investment amount, and years. Display the future amount in a text field when the user clicks the *Calculate* button, as shown in Figure 16.21(a).



(a) (b) (c)

FIGURE 16.21 (a) The user enters the investment amount, years, and interest rate to compute future value. (b) Exercise 16.8 displays the mouse position. (c) Exercise 16.9 uses the arrow keys to draw the lines.

Section 16.10

16.6** (*Alternating two messages*) Write a program to rotate with a mouse click two messages displayed on a panel, "Java is fun" and "Java is powerful".

16.7* (*Setting background color using a mouse*) Write a program that displays the background color of a panel as black when the mouse is pressed and as white when the mouse is released.

16.8* (*Displaying the mouse position*) Write two programs, such that one displays the mouse position when the mouse is clicked (see Figure 16.21(b)) and the other displays the mouse position when the mouse is pressed and ceases to display it when the mouse is released.

Section 16.11

16.9* (*Drawing lines using the arrow keys*) Write a program that draws line segments using the arrow keys. The line starts from the center of the frame and draws

toward east, north, west, or south when the right-arrow key, up-arrow key, left-arrow key, or down-arrow key is clicked, as shown in Figure 16.21(c).

16.10** (*Entering and displaying a string*) Write a program that receives a string from the keyboard and displays it on a panel. The *Enter* key signals the end of a string. Whenever a new string is entered, it is displayed on the panel.

16.11* (*Displaying a character*) Write a program to get a character input from the keyboard and display the character where the mouse points.

Section 16.12

16.12** (*Displaying a running fan*) Listing 15.4, DrawArcs.java, displays a motionless fan. Write a program that displays a running fan.

16.13** (*Slide show*) Twenty-five slides are stored as image files (slide0.jpg, slide1.jpg, . . . , slide24.jpg) in the image directory downloadable along with the source code in the book. The size of each image is 800 × 600. Write a Java application that automatically displays the slides repeatedly. Each slide is shown for a second. The slides are displayed in order. When the last slide finishes, the first slide is redisplayed, and so on.

(*Hint*: Place a label in the frame and set a slide as an image icon in the label.)

16.14** (*Raising flag*) Write a Java program that animates raising a flag, as shown in Figure 16.1. (See §15.11, "Displaying Images," on how to display images.)



Video Note
Animate a rising flag

16.15** (*Racing car*) Write a Java program that simulates car racing, as shown in Figure 16.22(a). The car moves from left to right. When it hits the right end, it restarts from the left and continues the same process. You can use a timer to control animation. Redraw the car with a new base coordinates (x, y), as shown in Figure 16.22(b).



FIGURE 16.22 (a) Exercise 16.15 displays a moving car. (b) You can redraw a car with a new base point.

16.16* (*Displaying a flashing label*) Write a program that displays a flashing label.

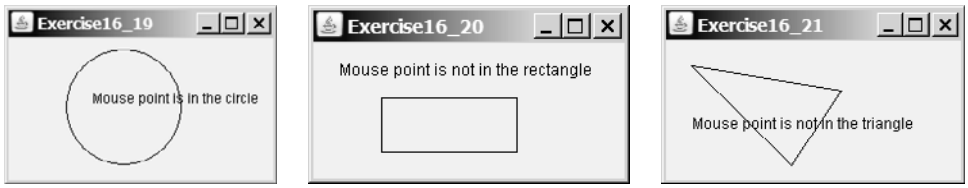
(*Hint*: To make the label flash, you need to repaint the panel alternately with the label and without it (blank screen) at a fixed rate. Use a `boolean` variable to control the alternation.)

16.17* (*Controlling a moving label*) Modify Listing 16.11, AnimationDemo.java, to control a moving label using the mouse. The label freezes when the mouse is pressed, and moves again when the button is released.

Comprehensive

16.18* (*Moving a circle using keys*) Write a program that moves a circle up, down, left, or right using the arrow keys.

16.19** (*Geometry: inside a circle?*) Write a program that draws a fixed circle centered at (100, 60) with radius 50. Whenever a mouse is moved, display the message indicating whether the mouse point is inside the circle, as shown in Figure 16.23(a).



(a) (b) (c)

FIGURE 16.23 Detect whether a point is inside a circle, a rectangle, or a triangle.

16.20** (*Geometry: inside a rectangle?*) Write a program that draws a fixed rectangle centered at (100, 60) with width 100 and height 40. Whenever a mouse is moved, display the message indicating whether the mouse point is inside the rectangle, as shown in Figure 16.23(b). To detect whether a point is inside a rectangle, use the `MyRectangle2D` class defined in Exercise 10.12.

16.21** (*Geometry: inside a triangle?*) Write a program that draws a fixed triangle with three vertices at (20, 20), (100, 100), and (140, 40). Whenever a mouse is moved, display the message indicating whether the mouse point is inside the triangle, as shown in Figure 16.23(c). To detect whether a point is inside a triangle, use the `Triangle2D` class defined in Exercise 10.13.

16.22*** (*Game: bean-machine animation*) Write a program that animates a bean machine introduced in Exercise 15.24. The animation terminates after ten balls are dropped, as shown in Figure 16.24.

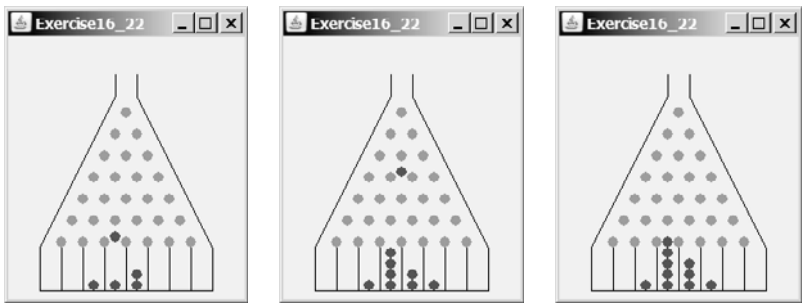


FIGURE 16.24 The balls are dropped to the bean machine.

16.23*** (*Geometry: closest pair of points*) Write a program that lets the user click on the panel to dynamically create points. Initially, the panel is empty. When a panel has two or more points, highlight the pair of closest points. Whenever a new point is created, a new pair of closest points is highlighted. Display the points using small circles and highlight the points using filled circles, as shown in Figure 16.25(a)–(c).

(*Hint*: store the points in an `ArrayList`.)

16.24* (*Controlling a clock*) Modify Listing 16.12 ClockAnimation.java to add two methods `start()` and `stop()` to start and stop the clock. Write a program

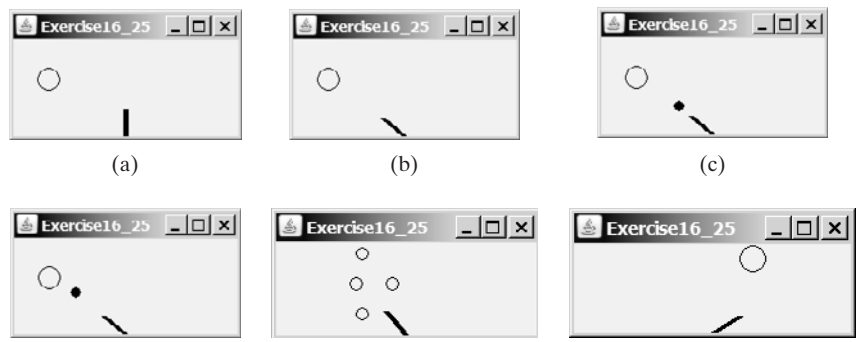


(a) (b) (c) (d)

FIGURE 16.25 Exercise 16.23 allows the user to create new points with a mouse click and highlights the pair of the closest points. Exercise 16.24 allows the user to start and stop a clock.

that lets the user control the clock with the *Start* and *Stop* buttons, as shown in Figure 16.25(d).

16.25* (*Game: hitting balloons*) Write a program that displays a balloon in a random position in a panel (Figure 16.26(a)). Use the left- and right-arrow keys to point the gun left or right to aim at the balloon (Figure 16.26(b)). Press the up-arrow key to fire a small ball from the gun (Figure 16.26(c)). Once the ball hits the balloon, the debris is displayed (Figure 16.26(e)) and a new balloon is displayed in a random location (Figure 16.26(f)). If the ball misses the balloon, the ball disappears once it hits the boundary of the panel. You can then press the up-arrow key to fire another ball. Whenever you press the left- or the right-arrow key, the gun turns 5 degrees left or right. (Instructors may modify the game as follows: 1. display the number of the balloons destroyed; 2. display a countdown timer (e.g., 60 seconds) and terminate the game once the time expires; 3. allow the balloon to rise dynamically.)



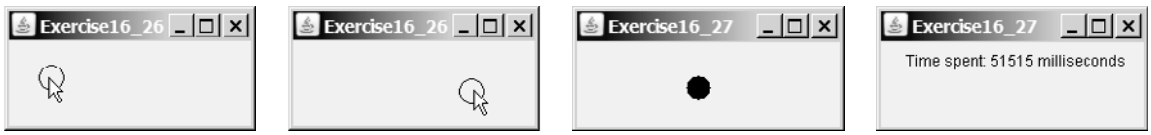
(a) (b) (c)

(d) (e) (f)

FIGURE 16.26 (a) A balloon is displayed in a random location. (b) Press the left-/right-arrow keys to aim the balloon. (c) Press the up-arrow key to fire a ball. (d) The ball moves straight toward the balloon. (e) The ball hits the balloon. (f) A new balloon is displayed in a random position.

16.26** (*Moving a circle using mouse*) Write a program that displays a circle with radius 10 pixels. You can point the mouse inside the circle and drag (i.e., move with mouse pressed) the circle wherever the mouse goes, as shown in Figure 16.27(a)–(b).

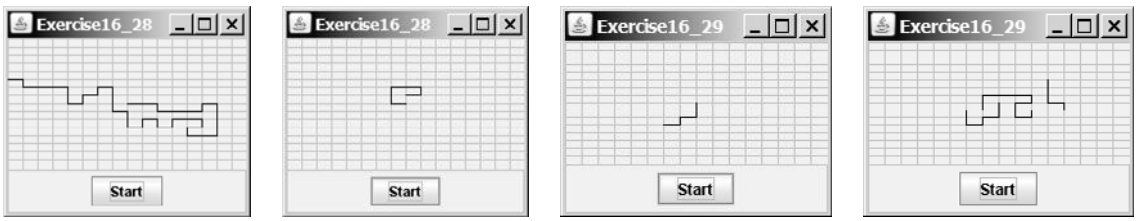
16.27*** (*Game: eye-hand coordination*) Write a program that displays a circle of radius 10 pixels filled with a random color at a random location on a panel, as shown in Figure 16.27(c). When you click the circle, it is gone and a new random-color circle is displayed at another random location. After twenty circles are clicked, display the time spent in the panel, as shown in Figure 16.27(d).



(a) (b) (c) (d)

FIGURE 16.27 (a)–(b) You can point, drag, and move the circle. (c) When you click a circle, a new circle is displayed at a random location. (d) After 20 circles are clicked, the time spent in the panel is displayed.

16.28* (*Simulation: self-avoiding random walk*) A self-avoiding walk in a lattice is a path from one point to another which does not visit the same point twice. Self-avoiding walks have applications in physics, chemistry, and mathematics. They can be used to model chainlike entities such as solvents and polymers. Write a program that displays a random path that starts from the center and ends at a point on the boundary, as shown in Figure 16.28(a), or ends at a dead-end point (i.e., surrounded by four points that are already visited), as shown in Figure 16.28(b). Assume the size of the lattice is **16** by **16**.



(a) (b) (c) (d)

FIGURE 16.28 (a) A path ends at a boundary point. (b) A path ends at dead-end point. (c)–(d) Animation shows the progress of a path step by step.

16.29* (*Animation: self-avoiding random walk*) Revise the preceding exercise to display the walk step by step in an animation, as shown in Figure 16.28(c)–(d).

16.30** (*Simulation: self-avoiding random walk*) Write a simulation program to show that the chance of getting dead-end paths increases as the grid size increases. Your program simulates lattices with size from 10 to 80. For each lattice size, simulate a self-avoiding random walk 10000 times and display the probability of the dead-end paths, as shown in the following sample output:

```
For a lattice of size 10, the probability of dead-end paths is 10.6%
For a lattice of size 11, the probability of dead-end paths is 14.0%
...
For a lattice of size 80, the probability of dead-end paths is 99.5%
```



16.31* (*Geometry: displaying an n-sided regular polygon*) Exercise 15.25 created the `RegularPolygonPanel` for displaying an n-sided regular polygon. Write a program that displays a regular polygon and uses two buttons named *+1* and *−1* to increase or decrease the size of the polygon, as shown in Figure 16.29(a)–(b).

16.32** (*Geometry: adding and removing points*) Write a program that lets the user click on the panel to dynamically create and remove points. When the user right-click the mouse, a point is created and displayed at the mouse point, and

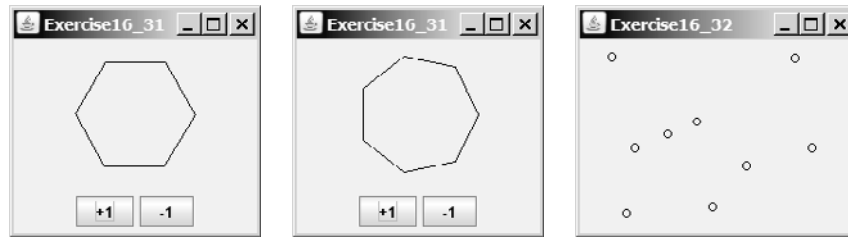


FIGURE 16.29 Clicking the *+1* or *−1* button increases or decreases the number of sides of a regular polygon in Exercise 16.31. Exercise 16.32 allows the user to create/remove points dynamically.

the user can remove a point by pointing to it and left-clicking the mouse, as shown in Figure 16.29(c).

16.33** (*Geometry: palindrome*) Write a program that animates a palindrome swing, as shown in Figure 16.30. Press the up-arrow key to increase the speed and the down-arrow key to decrease it. Press the *S* key to stop animation and the *R* key to resume.



FIGURE 16.30 Exercise 16.33 animates a palindrome swing.

16.34** (*Game: hangman*) Write a program that animates a hangman game swing, as shown in Figure 16.31. Press the up-arrow key to increase the speed and the down-arrow key to decrease it. Press the *S* key to stop animation and the *R* key to resume.

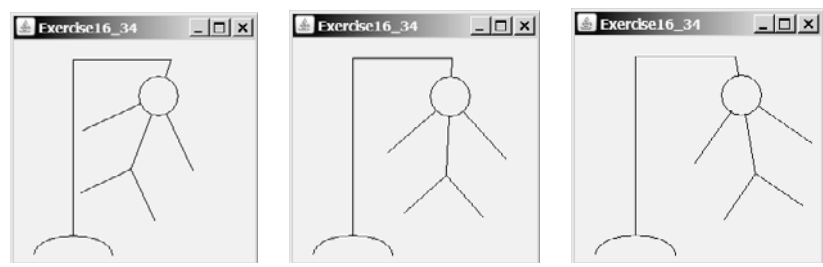
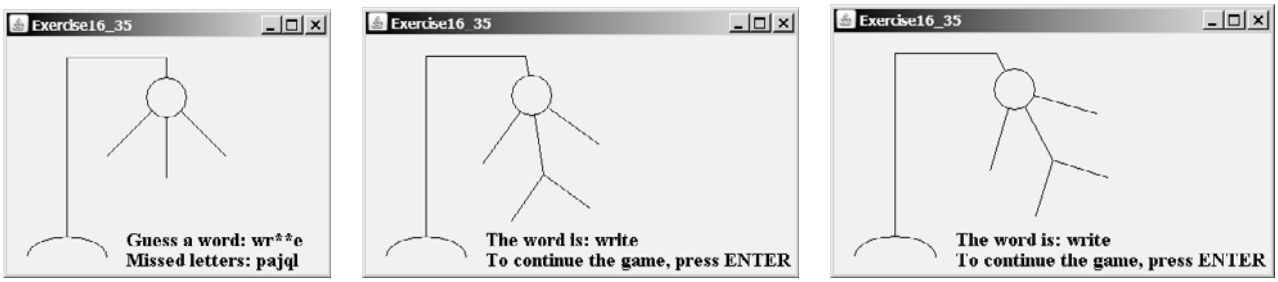


FIGURE 16.31 Exercise 16.34 animates a hangman game.

16.35*** (*Game: hangman*) Exercise 9.31 presents a console version of the popular hangman game. Write a GUI program that lets a user play the game. The user guesses a word by entering one letter at a time, as shown in Figure 16.32(a). If the user misses seven times, a hanging man swings, as shown in Figure 16.32(b)–(c). Once a word is finished, the user can press the *Enter* key to continue to guess another word.



(a) (b) (c)

FIGURE 16.32 Exercise 16.35 develops a complete hangman game.

16.36* (*Flipping coins*) Write a program that displays head (`H`) or tail (`T`) for each of nine coins, as shown in Figure 16.33. When a cell is clicked, the coin is flipped. A cell is a `JLable`. Write a custom cell class that extends `JLable` with the mouse listener for handling the clicks. When the program starts, all cells initially display `H`.

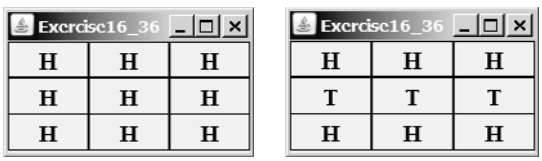


FIGURE 16.33 Exercise 16.36 enables the user to click a cell to flip a coin.

This page intentionally left blank

CHAPTER 17

CREATING GRAPHICAL USER INTERFACES

Objectives

- To create graphical user interfaces with various user-interface components: `JButton`, `JCheckBox`, `JRadioButton`, `JLabel`, `JTextField`, `JTextArea`, `JComboBox`, `JList`, `JScrollBar`, and `JSlider` (§§17.2–17.11).
- To create listeners for various types of events (§§17.2–17.11).
- To explore `JButton` (§17.2)
- To explore `JCheckBox` (§17.3)
- To explore `JRadioButton` (§17.4)
- To explore `JLabel` (§17.5)
- To explore `JTextField` (§17.6)
- To explore `JTextArea` (§17.7)
- To explore `JComboBox` (§17.8)
- To explore `JList` (§17.9)
- To explore `JScrollBar` (§17.10)
- To explore `JSlider` (§17.11)
- To display multiple windows in an application (§17.12).



17.1 Introduction

GUI

A graphical user interface (GUI) makes a system user friendly and easy to use. Creating a GUI requires creativity and knowledge of how GUI components work. Since the GUI components in Java are very flexible and versatile, you can create a wide assortment of useful user interfaces.

Many Java IDEs provide tools for visually designing and developing GUI interfaces. This enables you to rapidly assemble the elements of a user interface (UI) for a Java application or applet with minimum coding. Tools, however, cannot do everything. You have to modify the programs they produce. Consequently, before you begin to use the visual tools, you must understand the basic concepts of Java GUI programming.

Previous chapters briefly introduced several GUI components. This chapter introduces the frequently used GUI components in detail (see Figure 17.1). (Since this chapter introduces no new concepts, instructors may assign it for students to study on their own.)

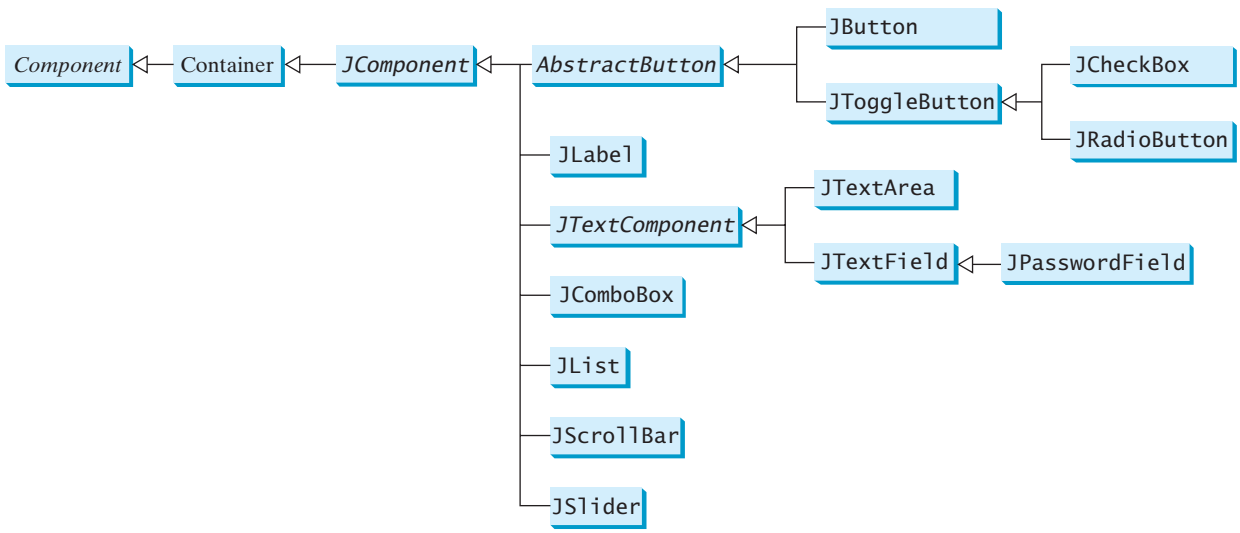


FIGURE 17.1 These Swing GUI components are frequently used to create user interfaces.



Note

naming convention for components

Throughout this book, the prefixes `jbt`, `jchk`, `jrb`, `jlbl`, `jtf`, `jpf`, `jta`, `jcbo`, `jlst`, `jscb`, and `jsld` are used to name reference variables for `JButton`, `JCheckBox`, `JRadioButton`, `JLabel`, `JTextField`, `JPasswordField`, `JTextArea`, `JComboBox`, `JList`, `JScrollBar`, and `JSlider`.

17.2 Buttons

A *button* is a component that triggers an action event when clicked. Swing provides regular buttons, toggle buttons, check box buttons, and radio buttons. The common features of these buttons are defined in `javax.swing.AbstractButton`, as shown in Figure 17.2.

`AbstractButton`
`JButton`

This section introduces the regular buttons defined in the `JButton` class. `JButton` inherits `AbstractButton` and provides several constructors to create buttons, as shown in Figure 17.3.

17.2.1 Icons, Pressed Icons, and Rollover Icons

A regular button has a default icon, a pressed icon, and a rollover icon. Normally you use the default icon. The other icons are for special effects. A pressed icon is displayed when a button is pressed, and a rollover icon is displayed when the mouse is over the button but not pressed.

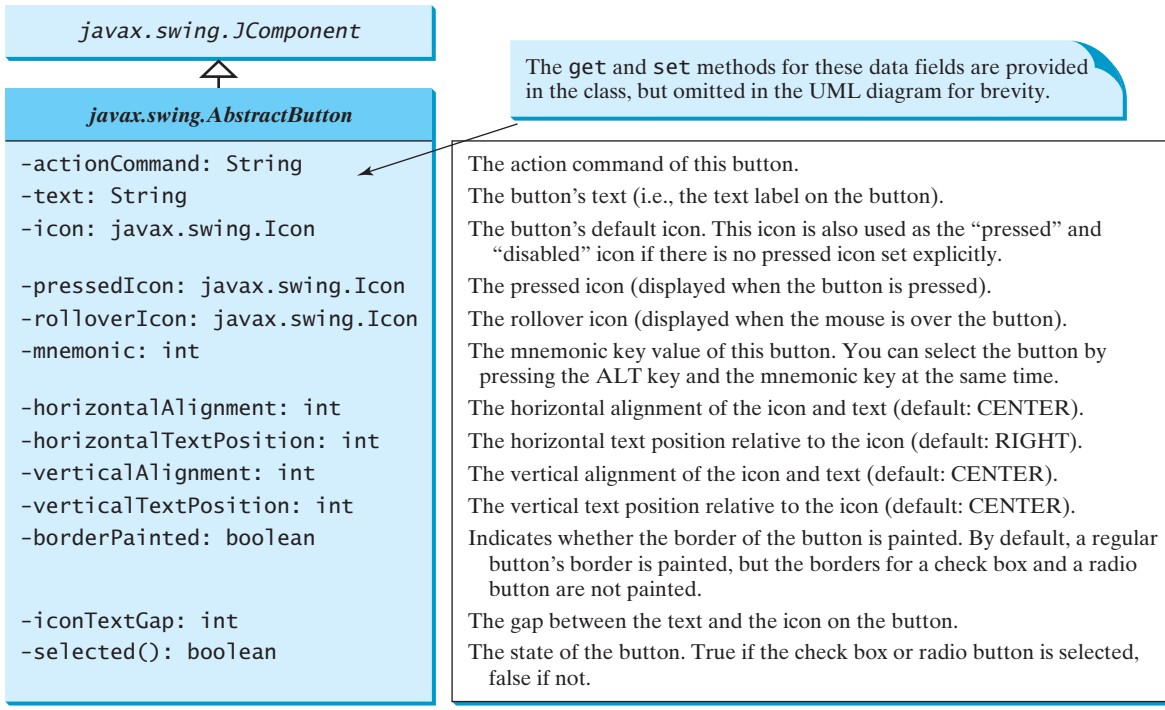

javax.swing.JComponent
javax.swing.AbstractButton
-actionCommand: String
-text: String
-icon: javax.swing.Icon
-pressedIcon: javax.swing.Icon
-rolloverIcon: javax.swing.Icon
-mnemonic: int
-horizontalAlignment: int
-horizontalTextPosition: int
-verticalAlignment: int
-verticalTextPosition: int
-borderPainted: boolean
-iconTextGap: int
-selected(): boolean
The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
The action command of this button.
The button's text (i.e., the text label on the button).
The button's default icon. This icon is also used as the "pressed" and "disabled" icon if there is no pressed icon set explicitly.
The pressed icon (displayed when the button is pressed).
The rollover icon (displayed when the mouse is over the button).
The mnemonic key value of this button. You can select the button by pressing the ALT key and the mnemonic key at the same time.
The horizontal alignment of the icon and text (default: CENTER).
The horizontal text position relative to the icon (default: RIGHT).
The vertical alignment of the icon and text (default: CENTER).
The vertical text position relative to the icon (default: CENTER).
Indicates whether the border of the button is painted. By default, a regular button's border is painted, but the borders for a check box and a radio button are not painted.
The gap between the text and the icon on the button.
The state of the button. True if the check box or radio button is selected, false if not.

FIGURE 17.2 `AbstractButton` defines common features of different types of buttons.

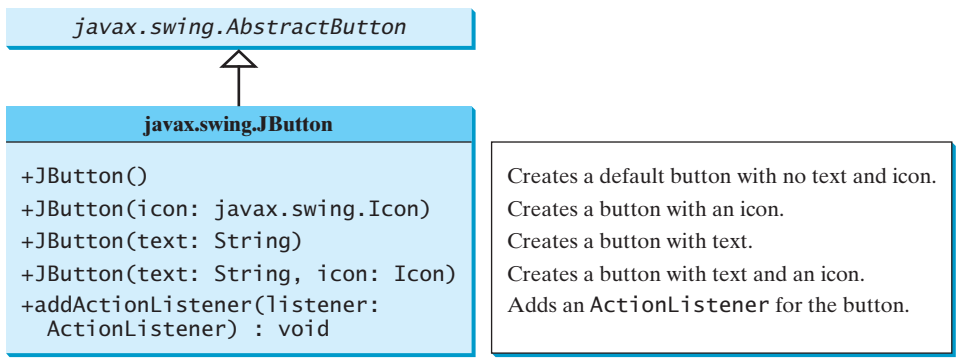

javax.swing.AbstractButton
javax.swing.JButton
+JButton()
+JButton(icon: javax.swing.Icon)
+JButton(text: String)
+JButton(text: String, icon: Icon)
+addActionListener(listener: ActionListener) : void
Creates a default button with no text and icon.
Creates a button with an icon.
Creates a button with text.
Creates a button with text and an icon.
Adds an ActionListener for the button.

FIGURE 17.3 `JButton` defines a regular push button.

For example, Listing 17.1 displays the American flag as a regular icon, the Canadian flag as a pressed icon, and the British flag as a rollover icon, as shown in Figure 17.4.

LISTING 17.1 TestButtonIcons.java

```
1  import javax.swing.*;
2
3  public class TestButtonIcons extends JFrame  {
4    public static void main(String[] args) {
5      // Create a frame and set its properties
6      JFrame frame = new TestButtonIcons();
7      frame.setTitle("ButtonIcons");
8      frame.setSize(200, 100);
9      frame.setLocationRelativeTo(null); // Center the frame
10     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
11      frame.setVisible(true);
12    }
13
14    public TestButtonIcons() {
15      ImageIcon usIcon = new ImageIcon("image/usIcon.gif");
16      ImageIcon caIcon = new ImageIcon("image/caIcon.gif");
17      ImageIcon ukIcon = new ImageIcon("image/ukIcon.gif");
18
19      JButton jbt = new JButton("Click it", usIcon);
20      jbt.setPressedIcon(caIcon);
21      jbt.setRolloverIcon(ukIcon);
22
23      add(jbt);
24    }
25  }
```

create icons

regular icon
pressed icon
rollover icon

add a button



(a) Default icon (b) Pressed icon (c) Rollover icon

FIGURE 17.4 A button can have several types of icons.

17.2.2 Alignments

horizontal alignment

Horizontal alignment specifies how the icon and text are placed horizontally on a button. You can set the horizontal alignment using `setHorizontalAlignment(int)` with one of the five constants `LEADING`, `LEFT`, `CENTER`, `RIGHT`, `TRAILING`, as shown in Figure 17.5. At present, `LEADING` and `LEFT` are the same, and `TRAILING` and `RIGHT` are the same. Future implementation may distinguish them. The default horizontal alignment is `SwingConstants.CENTER`.



Horizontally left Horizontally center Horizontally right

FIGURE 17.5 You can specify how the icon and text are placed on a button horizontally.

vertical alignment

Vertical alignment specifies how the icon and text are placed vertically on a button. You can set the vertical alignment using `setVerticalAlignment(int)` with one of the three constants `TOP`, `CENTER`, `BOTTOM`, as shown in Figure 17.6. The default vertical alignment is `SwingConstants.CENTER`.



Vertically top Vertically center Vertically bottom

FIGURE 17.6 You can specify how the icon and text are placed on a button vertically.

17.2.3 Text Positions

Horizontal text position specifies the horizontal position of the text relative to the icon. You can set the horizontal text position using `setHorizontalTextPosition(int)` with one of the five constants `LEADING`, `LEFT`, `CENTER`, `RIGHT`, `TRAILING`, as shown in Figure 17.7. At present, `LEADING` and `LEFT` are the same, and `TRAILING` and `RIGHT` are the same. Future implementation may distinguish them. The default horizontal text position is `SwingConstants.RIGHT`.

horizontal text position

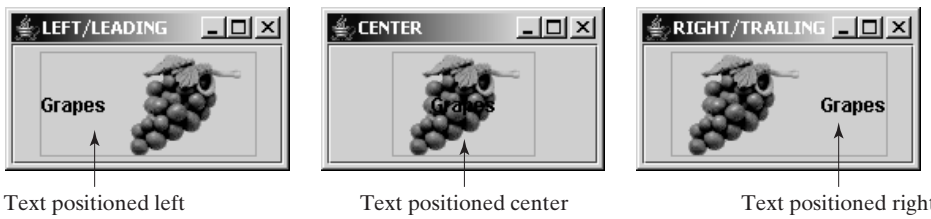


FIGURE 17.7 You can specify the horizontal position of the text relative to the icon.

Vertical text position specifies the vertical position of the text relative to the icon. You can set the vertical text position using `setVerticalTextPosition(int)` with one of the three constants `TOP`, `CENTER`, `BOTTOM`, as shown in Figure 17.8. The default vertical text position is `SwingConstants.CENTER`.

vertical text position

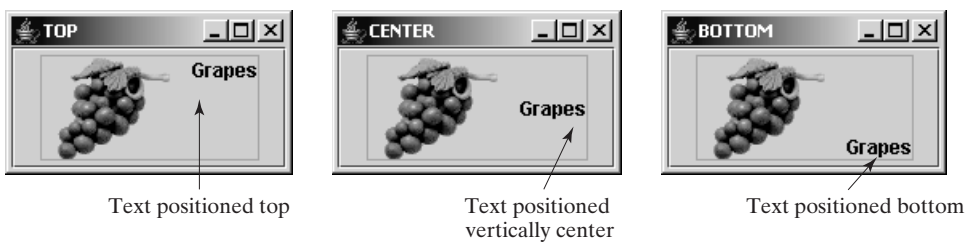


FIGURE 17.8 You can specify the vertical position of the text relative to the icon.



Note

The constants `LEFT`, `CENTER`, `RIGHT`, `LEADING`, `TRAILING`, `TOP`, and `BOTTOM` used in `AbstractButton` are also used in many other Swing components. These constants are centrally defined in the `javax.swing.SwingConstants` interface. Since all Swing GUI components implement `SwingConstants`, you can reference the constants through `SwingConstants` or a GUI component. For example, `SwingConstants.CENTER` is the same as `JButton.CENTER`.

SwingConstants

`JButton` can fire many types of events, but often you need to add listeners to respond to an `ActionEvent`. When a button is pressed, it fires an `ActionEvent`.

17.2.4 Using Buttons

This section presents a program, shown in Listing 17.2, that displays a message on a panel and uses two buttons, `<=` and `=>`, to move the message on the panel to the left or right. The layout of the UI is shown in Figure 17.9.



Video Note
Use buttons

Here are the major steps in the program:

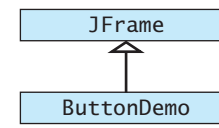
1. Create the user interface.
   Create a `MessagePanel` object to display the message. The `MessagePanel` class was created in Listing 15.8, MessagePanel.java. Place it in the center of the frame. Create two buttons, `<=` and `=>`, on a panel. Place the panel in the south of the frame.

2. Process the event.
   Create and register listeners for processing the action event to move the message left or right according to whether the left or right button was clicked.



FIGURE 17.9 Clicking the <= and => buttons causes the message on the panel to move to the left and right, respectively.

LISTING 17.2 ButtonDemo.java




```
1  import java.awt.*;
2  import java.awt.event.ActionListener;
3  import java.awt.event.ActionEvent;
4  import javax.swing.*;
5
6  public class ButtonDemo extends JFrame {
7    // Create a panel for displaying message
8    protected MessagePanel messagePanel
9      = new MessagePanel("Welcome to Java");
10
11   // Declare two buttons to move the message left and right
12   private JButton jbtLeft = new JButton("<=");
13   private JButton jbtRight = new JButton("=>");
14
15   public static void main(String[] args) {
16     ButtonDemo frame = new ButtonDemo();
17     frame.setTitle("ButtonDemo");
18     frame.setSize(250, 100);
19     frame.setLocationRelativeTo(null); // Center the frame
20     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21     frame.setVisible(true);
22   }
23
24   public ButtonDemo() {
25     // Set the background color of messagePanel
26     messagePanel.setBackground(Color.white);
27
28     // Create Panel jpButtons to hold two Buttons "<=" and "right =>"
29     JPanel jpButtons = new JPanel();
30     jpButtons.add(jbtLeft);
31     jpButtons.add(jbtRight);
32
33     // Set keyboard mnemonics
34     jbtLeft.setMnemonic('L');
35     jbtRight.setMnemonic('R');
36
37     // Set icons and remove text
38 //    jbtLeft.setIcon(new ImageIcon("image/left.gif"));
39 //    jbtRight.setIcon(new ImageIcon("image/right.gif"));
40 //    jbtLeft.setText(null);
```

create frame

create UI

mnemonic

```
41 //    jbtRight.setText(null);
42
43     // Set tool tip text on the buttons
44     jbtLeft.setToolTipText("Move message to left");                    tool tip
45     jbtRight.setToolTipText("Move message to right");
46
47     // Place panels in the frame
48     setLayout(new BorderLayout());
49     add(messagePanel, BorderLayout.CENTER);
50     add(jpButtons, BorderLayout.SOUTH);
51
52     // Register listeners with the buttons
53     jbtLeft.addActionListener(new ActionListener() {                   register listener
54       public void actionPerformed(ActionEvent e) {
55         messagePanel.moveLeft();
56       }
57     });
58     jbtRight.addActionListener(new ActionListener() {                  register listener
59       public void actionPerformed(ActionEvent e) {
60         messagePanel.moveRight();
61       }
62     });
63   }
64 }
```

`messagePanel` (line 8) is deliberately declared `protected` so that it can be referenced by a subclass in future examples.

You can set an icon image on the button by using the `setIcon` method. If you uncomment the following code in lines 38–41:

```
// jbtLeft.setIcon(new ImageIcon("image/left.gif"));
// jbtRight.setIcon(new ImageIcon("image/right.gif"));
// jbtLeft.setText(null);
// jbtRight.setText(null);
```

the texts are replaced by the icons, as shown in Figure 17.10(a). `"image/left.gif"` is located in `"c:\book\image\left.gif"`. Note that the backslash is the Windows file-path notation. In Java, the forward slash should be used.



(a) (b) (c)

FIGURE 17.10 You can set an icon on a `JButton` and access a button using its mnemonic key.

You can set text and an icon on a button at the same time, if you wish, as shown in Figure 17.10(b). By default, the text and icon are centered horizontally and vertically.

The button can also be accessed by using the keyboard mnemonics. Pressing *Alt*+*L* is equivalent to clicking the `<=` button, since you set the mnemonic property to `'L'` in the left button (line 34). If you change the left button text to `"Left"` and the right button text to `"Right"`, the `L` and `R` in the captions of these buttons will be underlined, as shown in Figure 17.10(b).

Each button has a tool tip text (lines 44–45), which appears when the mouse is set on the button without being clicked, as shown in Figure 17.10(c).

locating `MessagePanel`



Note

Since `MessagePanel` is not in the Java API, you should place MessagePanel.java in the same directory with ButtonDemo.java.

17.3 Check Boxes

toggle button

A *toggle button* is a two-state button like a light switch. `JToggleButton` inherits `AbstractButton` and implements a toggle button. Often `JToggleButton`'s subclasses `JCheckBox` and `JRadioButton` are used to enable the user to toggle a choice on or off. This section introduces `JCheckBox`. `JRadioButton` will be introduced in the next section.



Video Note
Use check boxes

`JCheckBox` inherits all the properties from `AbstractButton`, such as `text`, `icon`, `mnemonic`, `verticalAlignment`, `horizontalAlignment`, `horizontalTextPosition`, `verticalTextPosition`, and `selected`, and provides several constructors to create check boxes, as shown in Figure 17.11.

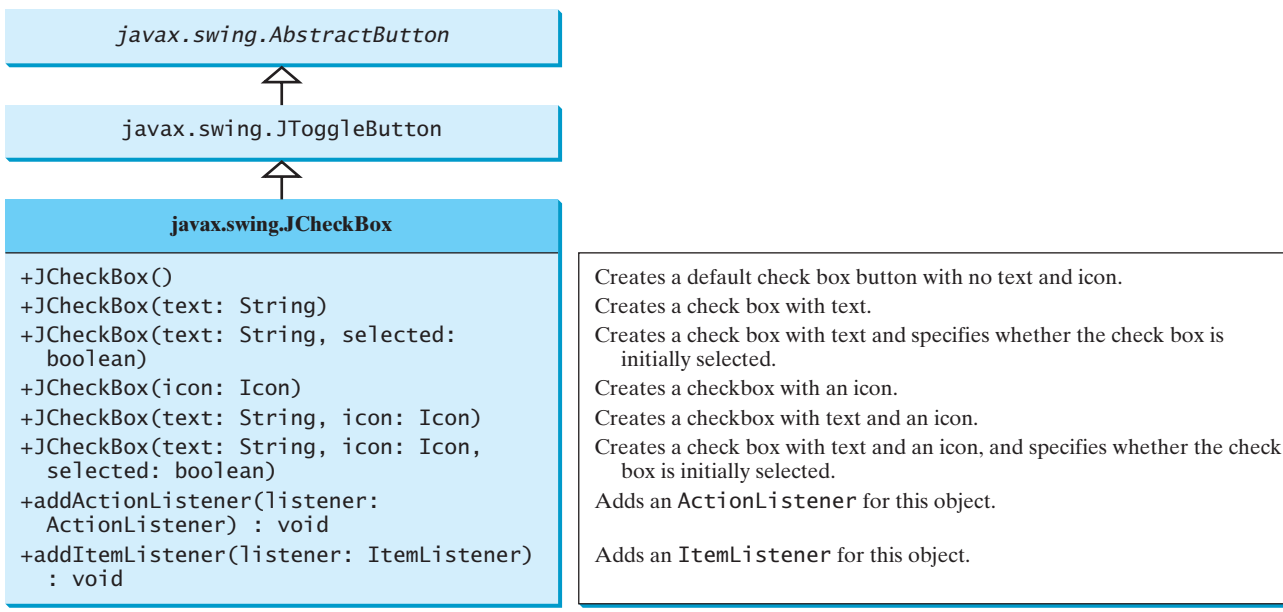
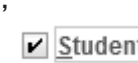

javax.swing.AbstractButton
javax.swing.JToggleButton
javax.swing.JCheckBox
+JCheckBox()
+JCheckBox(text: String)
+JCheckBox(text: String, selected: boolean)
+JCheckBox(icon: Icon)
+JCheckBox(text: String, icon: Icon)
+JCheckBox(text: String, icon: Icon, selected: boolean)
+addActionListener(listener: ActionListener) : void
+addItemListener(listener: ItemListener) : void
Creates a default check box button with no text and icon.
Creates a check box with text.
Creates a check box with text and specifies whether the check box is initially selected.
Creates a checkbox with an icon.
Creates a checkbox with text and an icon.
Creates a check box with text and an icon, and specifies whether the check box is initially selected.
Adds an ActionListener for this object.
Adds an ItemListener for this object.

FIGURE 17.11 `JCheckBox` defines a check box button.

Here is an example of a check box with text Student, foreground `red`, background `white`, mnemonic key `'S'`, and initially selected.

```
JCheckBox jchk = new JCheckBox("Student", true);
jchk.setForeground(Color.RED);
jchk.setBackground(Color.WHITE);
jchk.setMnemonic('S');
```



When a check box is clicked (checked or unchecked), it fires an `ItemEvent` and then an `ActionEvent`. To see if a check box is selected, use the `isSelected()` method.

Listing 17.3 gives a program that adds three check boxes named *Centered*, *Bold*, and *Italic* to the preceding example to let the user specify whether the message is centered, bold, or italic, as shown in Figure 17.12.

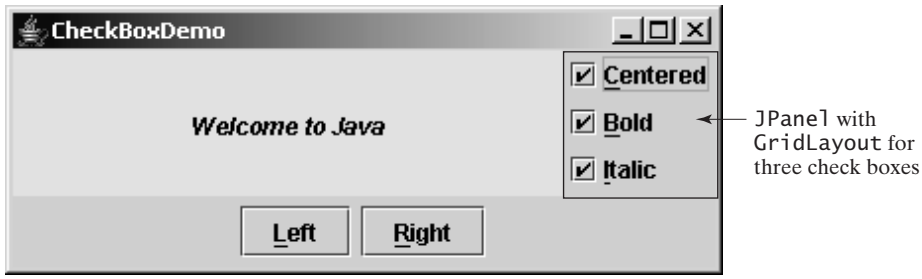
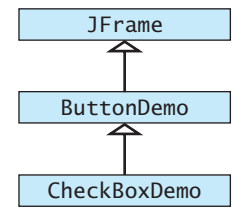


FIGURE 17.12 Three check boxes are added to specify how the message is displayed.

There are at least two approaches to writing this program. The first is to revise the preceding `ButtonDemo` class to insert the code for adding the check boxes and processing their events. The second is to create a subclass that extends `ButtonDemo`. Please implement the first approach as an exercise. Listing 17.3 gives the code to implement the second approach.

LISTING 17.3 CheckBoxDemo.java




```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class CheckBoxDemo extends ButtonDemo {
6    // Create three check boxes to control the display of message
7    private JCheckBox jchkCentered = new JCheckBox("Centered");
8    private JCheckBox jchkBold = new JCheckBox("Bold");
9    private JCheckBox jchkItalic = new JCheckBox("Italic");
10
11   public static void main(String[] args) {
12     CheckBoxDemo frame = new CheckBoxDemo();                    create frame
13     frame.setTitle("CheckBoxDemo");
14     frame.setSize(500, 200);
15     frame.setLocationRelativeTo(null); // Center the frame
16     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17     frame.setVisible(true);
18   }
19
20   public CheckBoxDemo() {                                       create UI
21     // Set mnemonic keys
22     jchkCentered.setMnemonic('C');
23     jchkBold.setMnemonic('B');
24     jchkItalic.setMnemonic('I');
25
26     // Create a new panel to hold check boxes
27     JPanel jpCheckBoxes = new JPanel();
28     jpCheckBoxes.setLayout(new GridLayout(3, 1));
29     jpCheckBoxes.add(jchkCentered);
30     jpCheckBoxes.add(jchkBold);
31     jpCheckBoxes.add(jchkItalic);
32     add(jpCheckBoxes, BorderLayout.EAST);
33
34     // Register listeners with the check boxes
35     jchkCentered.addActionListener(new ActionListener() {       register listener
36       public void actionPerformed(ActionEvent e) {
37         messagePanel.setCentered(jchkCentered.isSelected());
38       }
39     });
40     jchkBold.addActionListener(new ActionListener() {           register listener
```

```
41        public void actionPerformed(ActionEvent e) {
42          setNewFont();
43        }
44      });
45      jchkItalic.addActionListener(new ActionListener() {
46        public void actionPerformed(ActionEvent e) {
47          setNewFont();
48        }
49      });
50    }
51
52    private void setNewFont() {
53      // Determine a font style
54      int fontStyle = Font.PLAIN;
55      fontStyle += (jchkBold.isSelected() ? Font.BOLD : Font.PLAIN);
56      fontStyle += (jchkItalic.isSelected() ? Font.ITALIC : Font.PLAIN);
57
58      // Set font for the message
59      Font font = messagePanel.getFont();
60      messagePanel.setFont(
61        new Font(font.getName(), fontStyle, font.getSize()));
62    }
63  }
```

set a new font

CheckBoxDemo extends **ButtonDemo** and adds three check boxes to control how the message is displayed. When a **CheckBoxDemo** is constructed (line 12), its superclass's no-arg constructor is invoked, so you don't have to rewrite the code that is already in the constructor of **ButtonDemo**.

When a check box is checked or unchecked, the listener's **actionPerformed** method is invoked to process the event. When the *Centered* check box is checked or unchecked, the **centered** property of the **MessagePanel** class is set to **true** or **false**.

The current font name and size used in **MessagePanel** are obtained from **messagePanel.getFont()** using the **getName()** and **getSize()** methods. The font styles (**Font.BOLD** and **Font.ITALIC**) are specified in the check boxes. If no font style is selected, the font style is **Font.PLAIN**. Font styles are combined by adding together the selected integers representing the fonts.

The keyboard mnemonics *C*, *B*, and *I* are set on the check boxes *Centered*, *Bold*, and *Italic*, respectively (lines 22–24). You can use a mouse click or a shortcut key to select a check box.

The **setFont** method (line 60) defined in the **Component** class is inherited in the **MessagePanel** class. This method automatically invokes the **repaint** method. Invoking **setFont** in **messagePanel** automatically repaints the message.

A check box fires an **ActionEvent** and an **ItemEvent** when it is clicked. You could process either the **ActionEvent** or the **ItemEvent** to redisplay the message. The example processes the **ActionEvent**. If you wish to process the **ItemEvent**, create a listener for **ItemEvent** and register it with a check box. The listener must implement the **itemStateChanged** handler to process an **ItemEvent**. For example, the following code registers an **ItemListener** with **jchkCentered**:

```
// To listen for ItemEvent
jchkCentered.addItemListener(new ItemListener() {
  /** Handle ItemEvent */
  public void itemStateChanged(ItemEvent e) {
    messagePanel.setCentered(jchkCentered.isSelected());
  }
});
```

17.4 Radio Buttons



Video Note
Use radio buttons

Radio buttons, also known as *option buttons*, enable you to choose a single item from a group of choices. In appearance radio buttons resemble check boxes, but check boxes display a square that is either checked or blank, whereas radio buttons display a circle that is either filled (if selected) or blank (if not selected).

`JRadioButton` inherits `AbstractButton` and provides several constructors to create radio buttons, as shown in Figure 17.13. These constructors are similar to the constructors for `JCheckBox`.

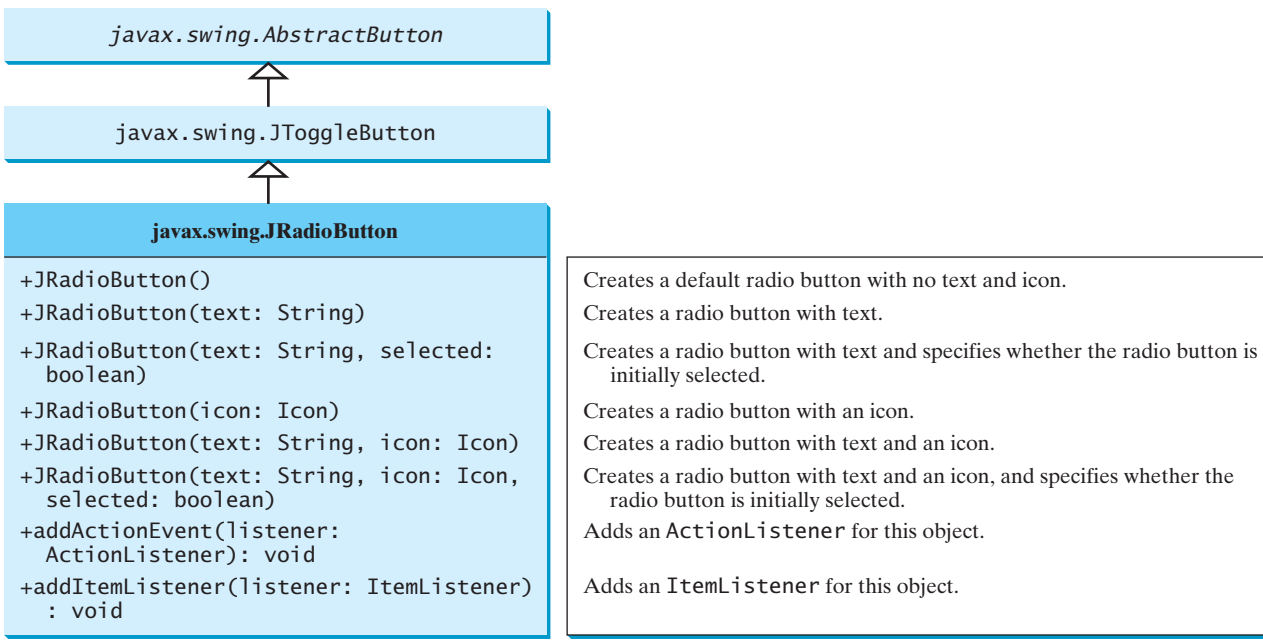
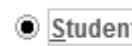

javax.swing.AbstractButton
javax.swing.JToggleButton
javax.swing.JRadioButton
+JRadioButton()
+JRadioButton(text: String)
+JRadioButton(text: String, selected: boolean)
+JRadioButton(icon: Icon)
+JRadioButton(text: String, icon: Icon)
+JRadioButton(text: String, icon: Icon, selected: boolean)
+addActionEvent(listener: ActionListener): void
+addItemListener(listener: ItemListener) : void
Creates a default radio button with no text and icon.
Creates a radio button with text.
Creates a radio button with text and specifies whether the radio button is initially selected.
Creates a radio button with an icon.
Creates a radio button with text and an icon.
Creates a radio button with text and an icon, and specifies whether the radio button is initially selected.
Adds an ActionListener for this object.
Adds an ItemListener for this object.

FIGURE 17.13 `JRadioButton` defines a radio button.

Here is an example of a radio button with text `Student`, `red` foreground, `white` background, mnemonic key `S`, and initially selected.

```
JRadioButton jrb = new JRadioButton("Student", true);
jrb.setForeground(Color.RED);
jrb.setBackground(Color.WHITE);
jrb.setMnemonic('S');
```



To group radio buttons, you need to create an instance of `java.swing.ButtonGroup` and use the `add` method to add them to it, as follows:

```
ButtonGroup group = new ButtonGroup();
group.add(jrb1);
group.add(jrb2);
```

This code creates a radio-button group for radio buttons `jrb1` and `jrb2` so that they are selected mutually exclusively. Without grouping, `jrb1` and `jrb2` would be independent.

GUI helper class

Note

`ButtonGroup` is not a subclass of `java.awt.Component`, so a `ButtonGroup` object cannot be added to a container.

When a radio button is changed (selected or deselected), it fires an `ItemEvent` and then an `ActionEvent`. To see if a radio button is selected, use the `isSelected()` method.

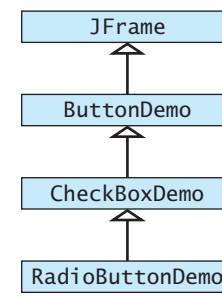
Listing 17.4 gives a program that adds three radio buttons named *Red*, *Green*, and *Blue* to the preceding example to let the user choose the color of the message, as shown in Figure 17.14.



FIGURE 17.14 Three radio buttons are added to specify the color of the message.

Again there are at least two approaches to writing this program. The first is to revise the preceding `CheckBoxDemo` class to insert the code for adding the radio buttons and processing their events. The second is to create a subclass that extends `CheckBoxDemo`. Listing 17.4 gives the code to implement the second approach.

LISTING 17.4 RadioButtonDemo.java




```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class RadioButtonDemo extends CheckBoxDemo {
6    // Declare radio buttons
7    private JRadioButton jrbRed, jrbGreen, jrbBlue;
8
9    public static void main(String[] args) {
10     RadioButtonDemo frame = new RadioButtonDemo();
11     frame.setSize(500, 200);
12     frame.setLocationRelativeTo(null); // Center the frame
13     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14     frame.setTitle("RadioButtonDemo");
15     frame.setVisible(true);
16   }
17
18   public RadioButtonDemo() {
19     // Create a new panel to hold check boxes
20     JPanel jpRadioButtons = new JPanel();
21     jpRadioButtons.setLayout(new GridLayout(3, 1));
22     jpRadioButtons.add(jrbRed = new JRadioButton("Red"));
23     jpRadioButtons.add(jrbGreen = new JRadioButton("Green"));
24     jpRadioButtons.add(jrbBlue = new JRadioButton("Blue"));
```

create frame

create UI

```
25      add(jpRadioButtons, BorderLayout.WEST);
26
27      // Create a radio-button group to group three buttons
28      ButtonGroup group = new ButtonGroup();                    group buttons
29      group.add(jrbRed);
30      group.add(jrbGreen);
31      group.add(jrbBlue);
32
33      // Set keyboard mnemonics
34      jrbRed.setMnemonic('E');
35      jrbGreen.setMnemonic('G');
36      jrbBlue.setMnemonic('U');
37
38      // Register listeners for radio buttons
39      jrbRed.addActionListener(new ActionListener() {           register listener
40        public void actionPerformed(ActionEvent e) {
41          messagePanel.setForeground(Color.red);
42        }
43      });
44      jrbGreen.addActionListener(new ActionListener() {         register listener
45        public void actionPerformed(ActionEvent e) {
46          messagePanel.setForeground(Color.green);
47        }
48      });
49      jrbBlue.addActionListener(new ActionListener() {          register listener
50        public void actionPerformed(ActionEvent e) {
51          messagePanel.setForeground(Color.blue);
52        }
53      });
54
55      // Set initial message color to blue
56      jrbBlue.setSelected(true);
57      messagePanel.setForeground(Color.blue);
58    }
59  }
```

RadioButtonDemo extends **CheckBoxDemo** and adds three radio buttons to specify the message color. When a radio button is clicked, its action event listener sets the corresponding foreground color in **messagePanel**.

The keyboard mnemonics **'R'** and **'B'** are already set for the *Right* button and *Bold* check box. To avoid conflict, the keyboard mnemonics **'E'**, **'G'**, and **'U'** are set on the radio buttons *Red*, *Green*, and *Blue*, respectively (lines 34–36).

The program creates a **ButtonGroup** and puts three **JRadioButton** instances (**jrbRed**, **jrbGreen**, and **jrbBlue**) in the group (lines 28–31).

A radio button fires an **ActionEvent** and an **ItemEvent** when it is selected or deselected. You could process either the **ActionEvent** or the **ItemEvent** to choose a color. The example processes the **ActionEvent**. Please rewrite the code using the **ItemEvent** as an exercise.

17.5 Labels

A *label* is a display area for a short text, an image, or both. It is often used to label other components (usually text fields). Figure 17.15 lists the constructors and methods in **JLabel**.

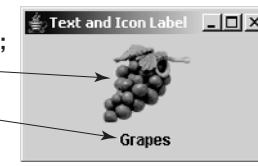
JLabel inherits all the properties from **JComponent** and has many properties similar to the ones in **JButton**, such as **text**, **icon**, **horizontalAlignment**, **verticalAlignment**,

horizontalTextPosition, **verticalTextPosition**, and **iconTextGap**. For example, the following code displays a label with text and an icon:

```
// Create an image icon from an image file
ImageIcon icon = new ImageIcon("image/grapes.gif");

// Create a label with a text, an icon,
// with centered horizontal alignment
JLabel jlbl = new JLabel("Grapes", icon, SwingConstants.CENTER);

//Set label's text alignment and gap between text and icon
jlbl.setHorizontalTextPosition(SwingConstants.CENTER);
jlbl.setVerticalTextPosition(SwingConstants.BOTTOM);
jlbl.setIconTextGap(5);
```



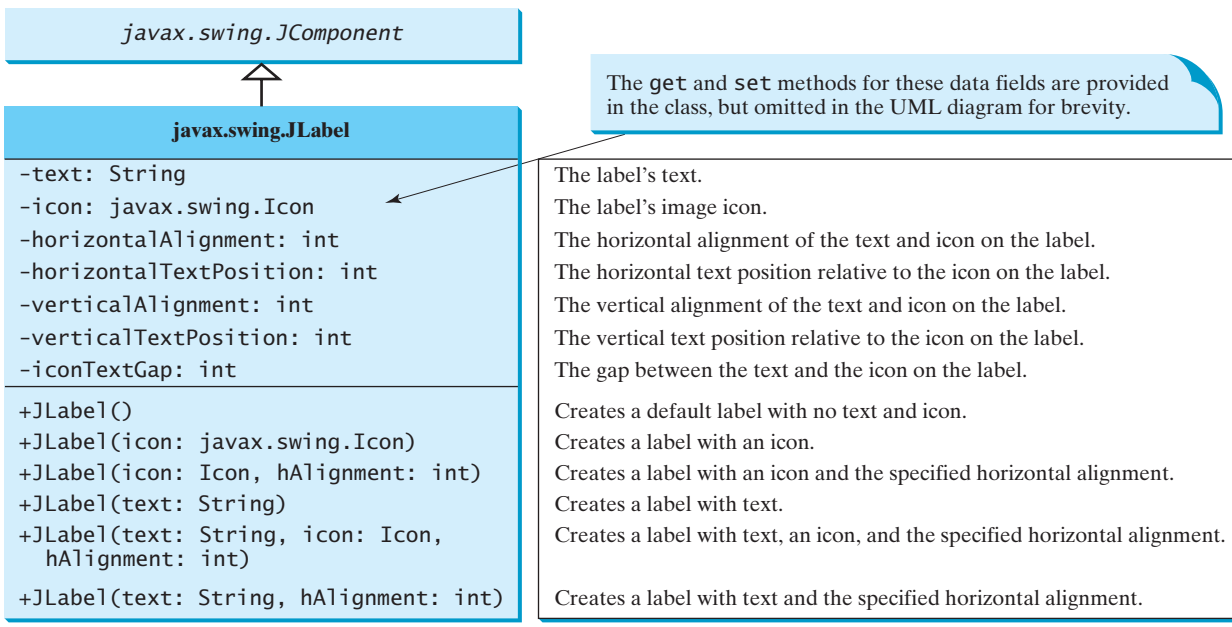

javax.swing.JComponent
javax.swing.JLabel
-text: String
-icon: javax.swing.Icon
-horizontalAlignment: int
-horizontalTextPosition: int
-verticalAlignment: int
-verticalTextPosition: int
-iconTextGap: int
+JLabel()
+JLabel(icon: javax.swing.Icon)
+JLabel(icon: Icon, hAlignment: int)
+JLabel(text: String)
+JLabel(text: String, icon: Icon, hAlignment: int)
+JLabel(text: String, hAlignment: int)
The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
The label's text.
The label's image icon.
The horizontal alignment of the text and icon on the label.
The horizontal text position relative to the icon on the label.
The vertical alignment of the text and icon on the label.
The vertical text position relative to the icon on the label.
The gap between the text and the icon on the label.
Creates a default label with no text and icon.
Creates a label with an icon.
Creates a label with an icon and the specified horizontal alignment.
Creates a label with text.
Creates a label with text, an icon, and the specified horizontal alignment.
Creates a label with text and the specified horizontal alignment.

FIGURE 17.15 **JLabel** displays text or an icon, or both.

17.6 Text Fields



Video Note
Use labels and text fields

A *text field* can be used to enter or display a string. **JTextField** is a subclass of **JTextComponent**. Figure 17.16 lists the constructors and methods in **JTextField**.

JTextField inherits **JTextComponent**, which inherits **JComponent**. Here is an example of creating a text field with red foreground color and right horizontal alignment:

```
JTextField jtfMessage = new JTextField("T-Strom");
jtfMessage.setForeground(Color.RED);
jtfMessage.setHorizontalAlignment(SwingConstants.RIGHT);
```

When you move the cursor in the text field and press the *Enter* key, it fires an **ActionEvent**.

Listing 17.5 gives a program that adds a text field to the preceding example to let the user set a new message, as shown in Figure 17.17.

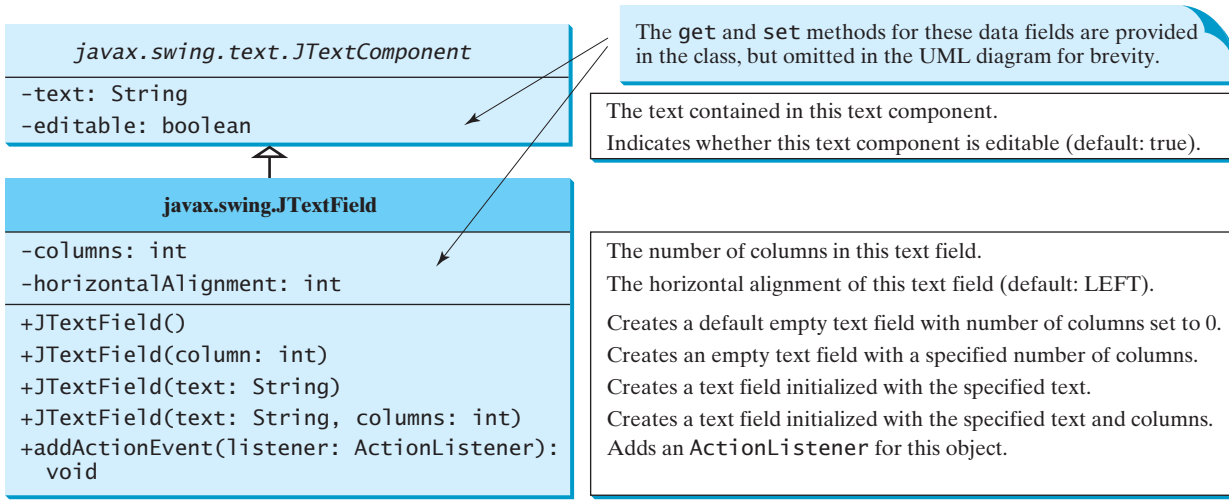
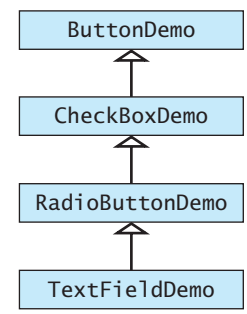

javax.swing.text.JTextComponent
-text: String
-editable: boolean
javax.swing.JTextField
-columns: int
-horizontalAlignment: int
+JTextField()
+JTextField(column: int)
+JTextField(text: String)
+JTextField(text: String, columns: int)
+addActionEvent(listener: ActionListener): void
The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
The text contained in this text component.
Indicates whether this text component is editable (default: true).
The number of columns in this text field.
The horizontal alignment of this text field (default: LEFT).
Creates a default empty text field with number of columns set to 0.
Creates an empty text field with a specified number of columns.
Creates a text field initialized with the specified text.
Creates a text field initialized with the specified text and columns.
Adds an ActionListener for this object.

FIGURE 17.16 `JTextField` enables you to enter or display a string.



FIGURE 17.17 A label and a text field are added to set a new message.

LISTING 17.5 TextFieldDemo.java




```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class TextFieldDemo extends RadioButtonDemo {
6    private JTextField jtfMessage = new JTextField(10);
7
8    /** Main method */
9    public static void main(String[] args) {
10     TextFieldDemo frame = new TextFieldDemo();
11     frame.pack();
12     frame.setTitle("TextFieldDemo");
13     frame.setLocationRelativeTo(null); // Center the frame
14     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15     frame.setVisible(true);
16   }
17
18   public TextFieldDemo() {
19     // Create a new panel to hold label and text field
20     JPanel jpTextField = new JPanel();
21     jpTextField.setLayout(new BorderLayout(5, 0));
```

create frame
pack frame

create UI

```
22      jpTextField.add(
23        new JLabel("Enter a new message"), BorderLayout.WEST);
24      jpTextField.add(jtfMessage, BorderLayout.CENTER);
25      add(jpTextField, BorderLayout.NORTH);
26
27      jtfMessage.setHorizontalAlignment(JTextField.RIGHT);
28
29      // Register listener
30      jtfMessage.addActionListener(new ActionListener() {
31        /** Handle ActionEvent */
32        public void actionPerformed(ActionEvent e) {
33          messagePanel.setMessage(jtfMessage.getText());
34          jtfMessage.requestFocusInWindow();
35        }
36      });
37    }
38  }
```

listener

TextFieldDemo extends **RadioButtonDemo** and adds a label and a text field to let the user enter a new message. After you set a new message in the text field and press the *Enter* key, a new message is displayed. Pressing the *Enter* key on the text field triggers an action event. The listener sets a new message in **messagePanel** (line 33).

pack()

The **pack()** method (line 11) automatically sizes the frame according to the size of the components placed in it.

requestFocusInWindow()

The **requestFocusInWindow()** method (line 34) defined in the **Component** class requests the component to receive input focus. Thus, **jtfMessage.requestFocusInWindow()** requests the input focus on **jtfMessage**. You will see the cursor on **jtfMessage** after the **actionPerformed** method is invoked.



Note

JPasswordField

If a text field is used for entering a password, use **JPasswordField** to replace **JTextField**. **JPasswordField** extends **JTextField** and hides the input text with echo characters (e.g., ******). By default, the echo character is *. You can specify a new echo character using the **setEchoChar(char)** method.

17.7 Text Areas

If you want to let the user enter multiple lines of text, you have to create several instances of **JTextField**. A better alternative is to use **JTextArea**, which enables the user to enter multiple lines of text. Figure 17.18 lists the constructors and methods in **JTextArea**.

Like **JTextField**, **JTextArea** inherits **JTextComponent**, which contains the methods **getText**, **setText**, **isEditable**, and **setEditable**. Here is an example of creating a text area with **5** rows and **20** columns, line-wrapped on words, **red** foreground color, and **Courier** font, **bold**, **20** pixels.

```
JTextArea jtaNote = new JTextArea("This is a text area", 5, 20);
jtaNote.setLineWrap(true);
jtaNote.setWrapStyleWord(true);
jtaNote.setForeground(Color.red);
jtaNote.setFont(new Font("Courier", Font.BOLD, 20));
```

wrap line
wrap word

JTextArea does not handle scrolling, but you can create a **JScrollPane** object to hold an instance of **JTextArea** and let **JScrollPane** handle scrolling for **JTextArea**, as follows:

```
// Create a scroll pane to hold text area
JScrollPane scrollPane = new JScrollPane(jta = new JTextArea());
add(scrollPane, BorderLayout.CENTER);
```

Listing 17.7 gives a program that displays an image and a text in a label, and a text in a text area, as shown in Figure 17.19.

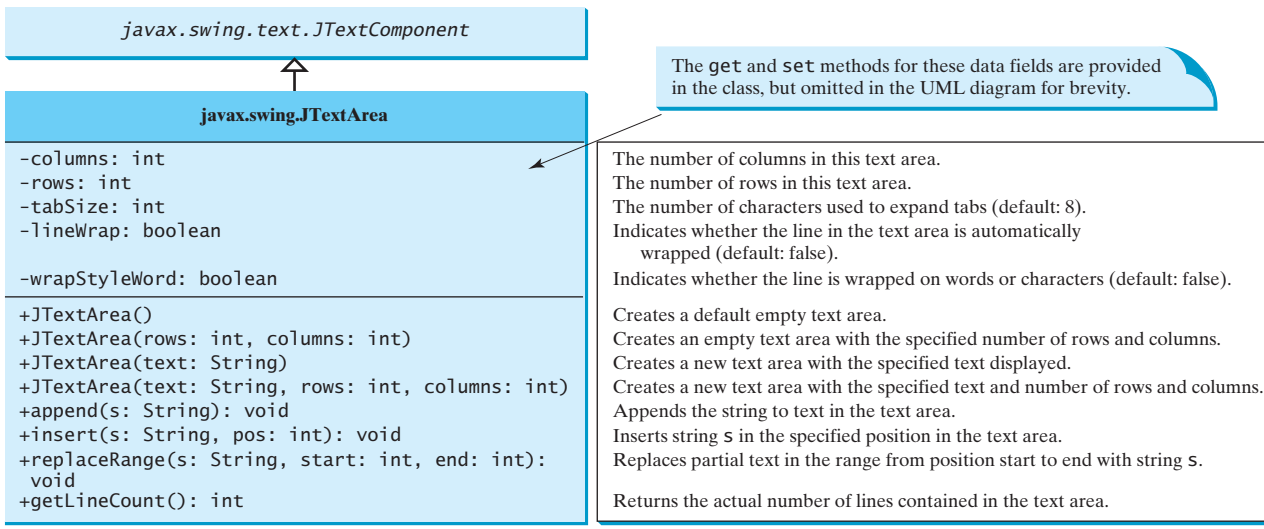

javax.swing.text.JTextComponent
javax.swing.JTextArea
-columns: int
-rows: int
-tabSize: int
-lineWrap: boolean
-wrapStyleWord: boolean
+JTextArea()
+JTextArea(rows: int, columns: int)
+JTextArea(text: String)
+JTextArea(text: String, rows: int, columns: int)
+append(s: String): void
+insert(s: String, pos: int): void
+replaceRange(s: String, start: int, end: int): void
+getLineCount(): int
The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
The number of columns in this text area.
The number of rows in this text area.
The number of characters used to expand tabs (default: 8).
Indicates whether the line in the text area is automatically wrapped (default: false).
Indicates whether the line is wrapped on words or characters (default: false).
Creates a default empty text area.
Creates an empty text area with the specified number of rows and columns.
Creates a new text area with the specified text displayed.
Creates a new text area with the specified text and number of rows and columns.
Appends the string to text in the text area.
Inserts string s in the specified position in the text area.
Replaces partial text in the range from position start to end with string s.
Returns the actual number of lines contained in the text area.

FIGURE 17.18 `JTextArea` enables you to enter or display multiple lines of characters.


TextAreaDemo
Canada
The Maple Leaf flag
The Canadian National Flag was adopted by the Canadian Parliament on October 22, 1964 and was proclaimed into law by
A label showing an image and a text
DescriptionPanel with BorderLayout
A text area inside a scroll panel

FIGURE 17.19 The program displays an image in a label, a title in a label, and a text in the text area.

Here are the major steps in the program:

1. Create a class named `DescriptionPanel` that extends `JPanel`, as shown in Listing 17.6. This class contains a text area inside a scroll pane, and a label for displaying an image icon and a title. This class is used in the present example and will be reused in later examples.

2. Create a class named `TextAreaDemo` that extends `JFrame`, as shown in Listing 17.7. Create an instance of `DescriptionPanel` and add it to the center of the frame. The relationship between `DescriptionPanel` and `TextAreaDemo` is shown in Figure 17.20.

LISTING 17.6 DescriptionPanel.java

```
1 import javax.swing.*;
2 import java.awt.*;
3
4 public class DescriptionPanel extends JPanel {
5   /** Label for displaying an image icon and a text */
6   private JLabel jlblImageTitle = new JLabel();                label
7
```

text area

label properties

wrap line
wrap word
read only

scroll pane

```
 8   /** Text area for displaying text */
 9   private JTextArea jtaDescription = new JTextArea();
10
11   public DescriptionPanel() {
12     // Center the icon and text and place the text under the icon
13     jlblImageTitle.setHorizontalAlignment(JLabel.CENTER);
14     jlblImageTitle.setHorizontalTextPosition(JLabel.CENTER);
15     jlblImageTitle.setVerticalTextPosition(JLabel.BOTTOM);
16
17     // Set the font in the label and the text field
18     jlblImageTitle.setFont(new Font("SansSerif", Font.BOLD, 16));
19     jtaDescription.setFont(new Font("Serif", Font.PLAIN, 14));
20
21     // Set lineWrap and wrapStyleWord true for the text area
22     jtaDescription.setLineWrap(true);
23     jtaDescription.setWrapStyleWord(true);
24     jtaDescription.setEditable(false);
25
26     // Create a scroll pane to hold the text area
27     JScrollPane scrollPane = new JScrollPane(jtaDescription);
28
29     // Set BorderLayout for the panel, add label and scrollpane
30     setLayout(new BorderLayout(5, 5));
31     add(scrollPane, BorderLayout.CENTER);
32     add(jlblImageTitle, BorderLayout.WEST);
33   }
34
35   /** Set the title */
36   public void setTitle(String title) {
37     jlblImageTitle.setText(title);
38   }
39
40   /** Set the image icon */
41   public void setImageIcon(ImageIcon icon) {
42     jlblImageTitle.setIcon(icon);
43   }
44
45   /** Set the text description */
46   public void setDescription(String text) {
47     jtaDescription.setText(text);
48   }
49 }
```

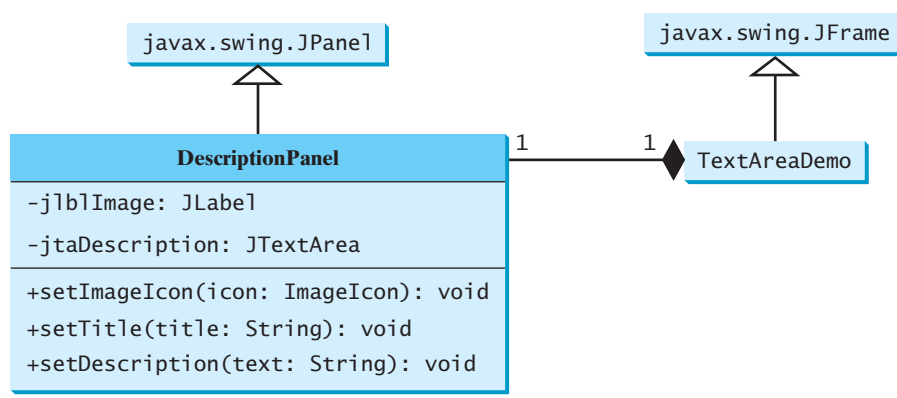


FIGURE 17.20 `TextAreaDemo` uses `DescriptionPanel` to display an image, title, and text description of a national flag.

The text area is inside a `JScrollPane` (line 27), which provides scrolling functions for the text area. Scroll bars automatically appear if there is more text than the physical size of the text area, and disappear if the text is deleted and the remaining text does not exceed the text area size.

The `lineWrap` property is set to `true` (line 22) so that the line is automatically wrapped when the text cannot fit in one line. The `wrapStyleWord` property is set to `true` (line 23) so that the line is wrapped on words rather than characters. The text area is set noneditable (line 24), so you cannot edit the description in the text area.

It is not necessary to create a separate class for `DescriptionPanel` in this example. Nevertheless, this class was created for reuse in the next section, where you will use it to display a description panel for various images.

LISTING 17.7 TextAreaDemo.java

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class TextAreaDemo extends JFrame {
5    // Declare and create a description panel
6    private DescriptionPanel descriptionPanel = new DescriptionPanel();    create decriptionPanel
7
8    public static void main(String[] args) {
9      TextAreaDemo frame = new TextAreaDemo();                             create frame
10     frame.pack();
11     frame.setLocationRelativeTo(null); // Center the frame
12     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13     frame.setTitle("TextAreaDemo");
14     frame.setVisible(true);
15   }
16
17   public TextAreaDemo() {
18     // Set title, text and image in the description panel                 create UI
19     descriptionPanel.setTitle("Canada");
20     String description = "The Maple Leaf flag \n\n" +
21       "The Canadian National Flag was adopted by the Canadian " +
22       "Parliament on October 22, 1964 and was proclaimed into law " +
23       "by Her Majesty Queen Elizabeth II (the Queen of Canada) on " +
24       "February 15, 1965. The Canadian Flag (colloquially known " +
25       "as The Maple Leaf Flag) is a red flag of the proportions " +
26       "two by length and one by width, containing in its center a " +
27       "white square, with a single red stylized eleven-point " +
28       "maple leaf centered in the white square.";
29     descriptionPanel.setDescription(description);
30     descriptionPanel.setImageIcon(new ImageIcon("image/ca.gif"));
31
32     // Add the description panel to the frame
33     setLayout(new BorderLayout());
34     add(descriptionPanel, BorderLayout.CENTER);                          add decriptionPanel
35   }
36 }
```

`TextAreaDemo` simply creates an instance of `DescriptionPanel` (line 6) and sets the title (line 19), image (line 30), and text in the description panel (line 29). `DescriptionPanel` is a subclass of `JPanel`. `DescriptionPanel` contains a label for displaying an image icon and a text title, and a text area for displaying a description of the image.

17.8 Combo Boxes

A *combo box*, also known as a *choice list* or *drop-down list*, contains a list of items from which the user can choose. It is useful in limiting a user's range of choices and avoids the cumbersome validation of data input. Figure 17.21 lists several frequently used constructors and methods in `JComboBox`.

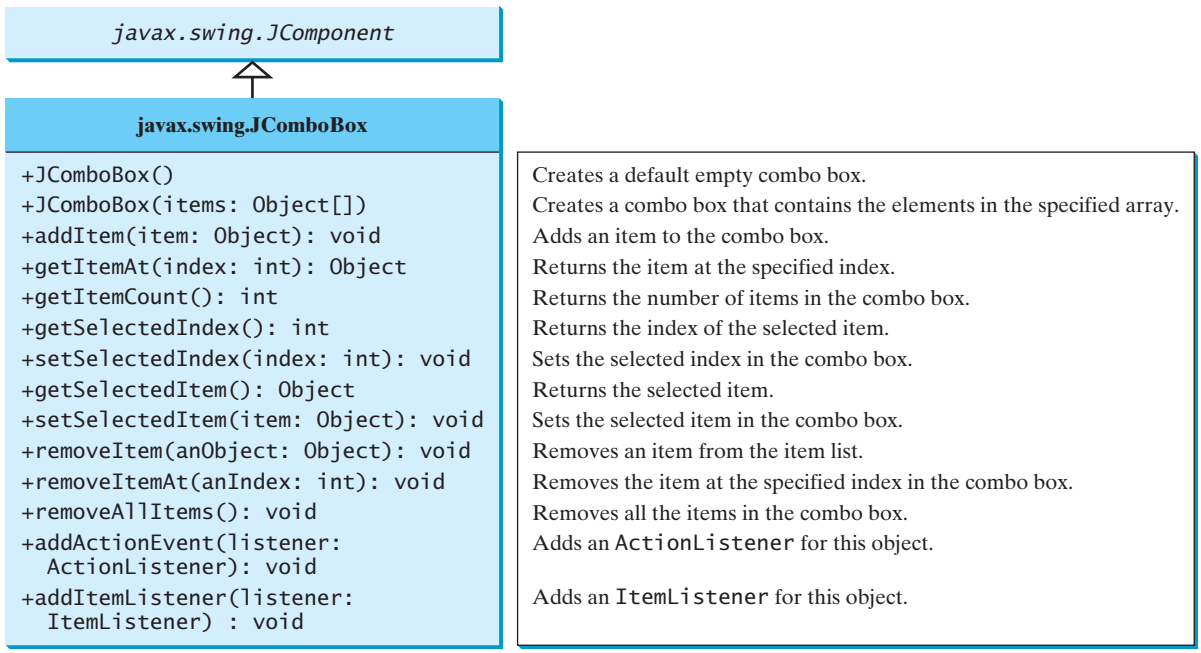
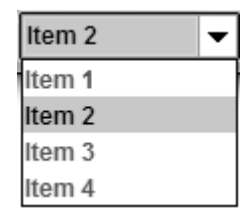

javax.swing.JComponent
javax.swing.JComboBox
+JComboBox()
+JComboBox(items: Object[])
+addItem(item: Object): void
+getItemAt(index: int): Object
+getItemCount(): int
+getSelectedIndex(): int
+setSelectedIndex(index: int): void
+getSelectedItem(): Object
+setSelectedItem(item: Object): void
+removeItem(anObject: Object): void
+removeItemAt(anIndex: int): void
+removeAllItems(): void
+addActionEvent(listener: ActionListener): void
+addItemListener(listener: ItemListener) : void
Creates a default empty combo box.
Creates a combo box that contains the elements in the specified array.
Adds an item to the combo box.
Returns the item at the specified index.
Returns the number of items in the combo box.
Returns the index of the selected item.
Sets the selected index in the combo box.
Returns the selected item.
Sets the selected item in the combo box.
Removes an item from the item list.
Removes the item at the specified index in the combo box.
Removes all the items in the combo box.
Adds an ActionListener for this object.
Adds an ItemListener for this object.

FIGURE 17.21 `JComboBox` enables you to select an item from a set of items.

The following statements create a combo box with four items, red foreground, white background, and the second item selected.

```
JComboBox jcb = new JComboBox(new Object[]
  {"Item 1", "Item 2", "Item 3", "Item 4"});
jcb.setForeground(Color.red);
jcb.setBackground(Color.white);
jcb.setSelectedItem("Item 2");
```



`JComboBox` can fire `ActionEvent` and `ItemEvent`, among many other events. Whenever an item is selected, an `ActionEvent` is fired. Whenever a new item is selected, `JComboBox` fires `ItemEvent` twice, once for deselecting the previously selected item, and the other for selecting the currently selected item. Note that no `ItemEvent` is fired if the current item is reselected. To respond to an `ItemEvent`, you need to implement the `itemStateChanged(ItemEvent e)` handler for processing a choice. To get data from a `JComboBox` menu, you can use `getSelectedItem()` to return the currently selected item, or `e.getItem()` method to get the item from the `itemStateChanged(ItemEvent e)` handler.

Listing 17.8 gives a program that lets users view an image and a description of a country's flag by selecting the country from a combo box, as shown in Figure 17.22.

Here are the major steps in the program:

1. Create the user interface.

   Create a combo box with country names as its selection values. Create a `DescriptionPanel` object. The `DescriptionPanel` class was introduced in the preceding

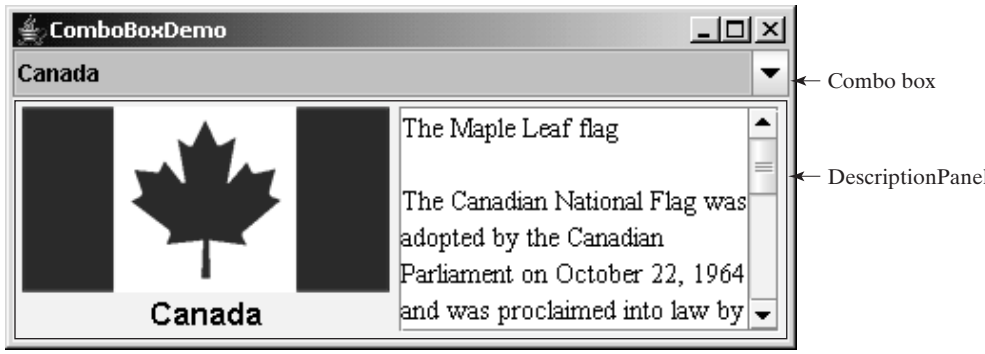

ComboBoxDemo
Canada
Combo box
The Maple Leaf flag
The Canadian National Flag was adopted by the Canadian Parliament on October 22, 1964 and was proclaimed into law by
DescriptionPanel
Canada

FIGURE 17.22 A country's info, including a flag image and a description of the flag, is displayed when the country is selected in the combo box.

example. Place the combo box in the north of the frame and the description panel in the center of the frame.

2. Process the event.
   Create a listener to implement the `itemStateChanged` handler to set the flag title, image, and text in the description panel for the selected country name.

LISTING 17.8 ComboBoxDemo.java

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class ComboBoxDemo extends JFrame {
6    // Declare an array of Strings for flag titles
7    private String[] flagTitles = {"Canada", "China", "Denmark",          country
8      "France", "Germany", "India", "Norway", "United Kingdom",
9      "United States of America"};
10
11   // Declare an ImageIcon array for the national flags of 9 countries
12   private ImageIcon[] flagImage = {                                     image icon
13     new ImageIcon("image/ca.gif"),
14     new ImageIcon("image/china.gif"),
15     new ImageIcon("image/denmark.gif"),
16     new ImageIcon("image/fr.gif"),
17     new ImageIcon("image/germany.gif"),
18     new ImageIcon("image/india.gif"),
19     new ImageIcon("image/norway.gif"),
20     new ImageIcon("image/uk.gif"),
21     new ImageIcon("image/us.gif")
22   };
23
24   // Declare an array of strings for flag descriptions
25   private String[] flagDescription = new String[9];                     description
26
27   // Declare and create a description panel
28   private DescriptionPanel descriptionPanel = new DescriptionPanel();
29
30   // Create a combo box for selecting countries
31   private JComboBox jcbo = new JComboBox(flagTitles);                   combo box
32
33   public static void main(String[] args) {
34     ComboBoxDemo frame = new ComboBoxDemo();
```

```
35      frame.pack();
36      frame.setTitle("ComboBoxDemo");
37      frame.setLocationRelativeTo(null); // Center the frame
38      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
39      frame.setVisible(true);
40    }
41
42    public ComboBoxDemo() {
43      // Set text description
44      flagDescription[0] = "The Maple Leaf flag \n\n" +
45        "The Canadian National Flag was adopted by the Canadian " +
46        "Parliament on October 22, 1964 and was proclaimed into law " +
47        "by Her Majesty Queen Elizabeth II (the Queen of Canada) on " +
48        "February 15, 1965. The Canadian Flag (colloquially known " +
49        "as The Maple Leaf Flag) is a red flag of the proportions " +
50        "two by length and one by width, containing in its center a " +
51        "white square, with a single red stylized eleven-point " +
52        "maple leaf centered in the white square.";
53      flagDescription[1] = "Description for China ... ";
54      flagDescription[2] = "Description for Denmark ... ";
55      flagDescription[3] = "Description for France ... ";
56      flagDescription[4] = "Description for Germany ... ";
57      flagDescription[5] = "Description for India ... ";
58      flagDescription[6] = "Description for Norway ... ";
59      flagDescription[7] = "Description for UK ... ";
60      flagDescription[8] = "Description for US ... ";
61
62      // Set the first country (Canada) for display
63      setDisplay(0);
64
65      // Add combo box and description panel to the list
create UI 66      add(jcbo, BorderLayout.NORTH);
67      add(descriptionPanel, BorderLayout.CENTER);
68
69      // Register listener
listener 70      jcbo.addItemListener(new ItemListener() {
71        /** Handle item selection */
72        public void itemStateChanged(ItemEvent e) {
73          setDisplay(jcbo.getSelectedIndex());
74        }
75      });
76    }
77
78    /** Set display information on the description panel */
79    public void setDisplay(int index) {
80      descriptionPanel.setTitle(flagTitles[index]);
81      descriptionPanel.setImageIcon(flagImage[index]);
82      descriptionPanel.setDescription(flagDescription[index]);
83    }
84  }
```

The listener listens to **ItemEvent** from the combo box and implements **ItemListener** (lines 70–75). Instead of using **ItemEvent**, you may rewrite the program to use **ActionEvent** for handling combo-box item selection.

The program stores the flag information in three arrays: **flagTitles**, **flagImage**, and **flagDescription** (lines 7–25). The array **flagTitles** contains the names of nine countries, the array **flagImage** contains images of the nine countries' flags, and the array **flagDescription** contains descriptions of the flags.

The program creates an instance of `DescriptionPanel` (line 28), which was presented in Listing 17.6, DescriptionPanel.java. The program creates a combo box with initial values from `flagTitles` (line 31). When the user selects an item in the combo box, the `itemStateChanged` handler is executed, finds the selected index, and sets its corresponding flag title, flag image, and flag description on the panel.

17.9 Lists

A *list* is a component that basically performs the same function as a combo box but enables the user to choose a single value or multiple values. The Swing `JList` is very versatile. Figure 17.23 lists several frequently used constructors and methods in `JList`.

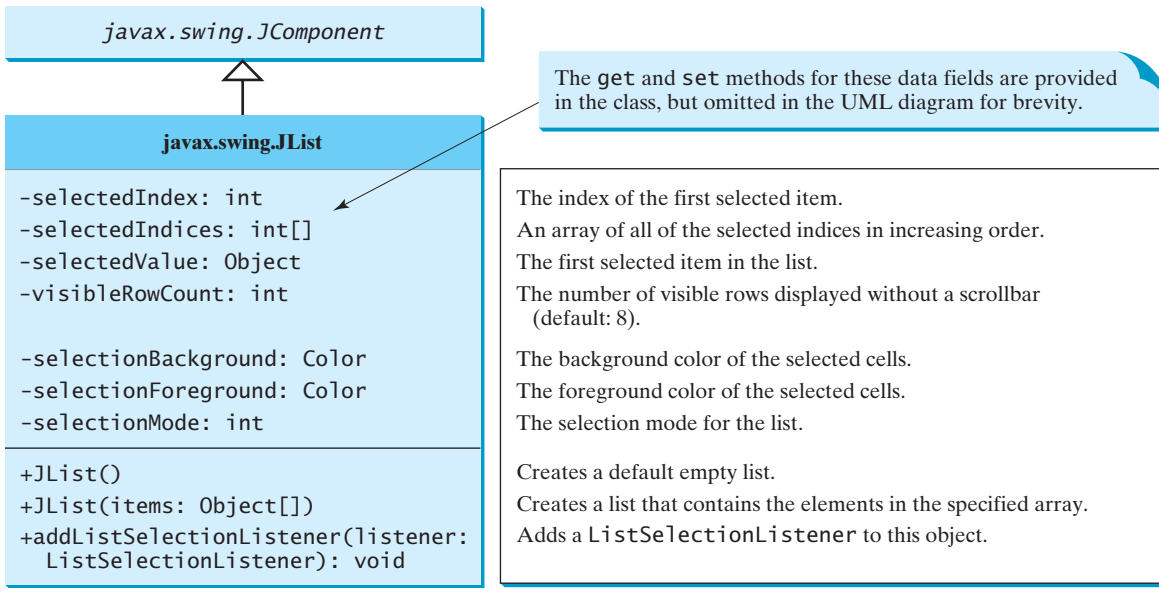

javax.swing.JComponent
javax.swing.JList
-selectedIndex: int
-selectedIndices: int[]
-selectedValue: Object
-visibleRowCount: int
-selectionBackground: Color
-selectionForeground: Color
-selectionMode: int
+JList()
+JList(items: Object[])
+addListSelectionListener(listener: ListSelectionListener): void
The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
The index of the first selected item.
An array of all of the selected indices in increasing order.
The first selected item in the list.
The number of visible rows displayed without a scrollbar (default: 8).
The background color of the selected cells.
The foreground color of the selected cells.
The selection mode for the list.
Creates a default empty list.
Creates a list that contains the elements in the specified array.
Adds a ListSelectionListener to this object.

FIGURE 17.23 `JList` enables you to select multiple items from a set of items.

`selectionMode` is one of the three values (`SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION`, `MULTIPLE_INTERVAL_SELECTION`) defined in `javax.swing.ListSelectionModel` that indicate whether a single item, single-interval item, or multiple-interval item can be selected. Single selection allows only one item to be selected. Single-interval selection allows multiple selections, but the selected items must be contiguous. Multiple-interval selection allows selections of multiple contiguous items without restrictions, as shown in Figure 17.24. The default value is `MULTIPLE_INTERVAL_SELECTION`.



(a) Single selection (b) Single-interval selection (c) Multiple-interval selection

FIGURE 17.24 `JList` has three selection modes: single selection, single-interval selection, and multiple-interval selection.

The following statements create a list with six items, `red` foreground, `white` background, `pink` selection foreground, `black` selection background, and visible row count `4`.

```
JList jlst = new JList(new Object[]
  {"Item 1", "Item 2", "Item 3", "Item 4", "Item 5", "Item 6"});
jlst.setForeground(Color.RED);
jlst.setBackground(Color.WHITE);
jlst.setSelectionForeground(Color.PINK);
jlst.setSelectionBackground(Color.BLACK);
jlst.setVisibleRowCount(4);
```

Lists do not scroll automatically. To make a list scrollable, create a scroll pane and add the list to it.

`JList` fires `javax.swing.event.ListSelectionEvent` to notify the listeners of the selections. The listener must implement the `valueChanged` handler in the `javax.swing.event.ListSelectionListener` interface to process the event.

Listing 17.9 gives a program that lets users select countries in a list and display the flags of the selected countries in the labels. Figure 17.25 shows a sample run of the program.

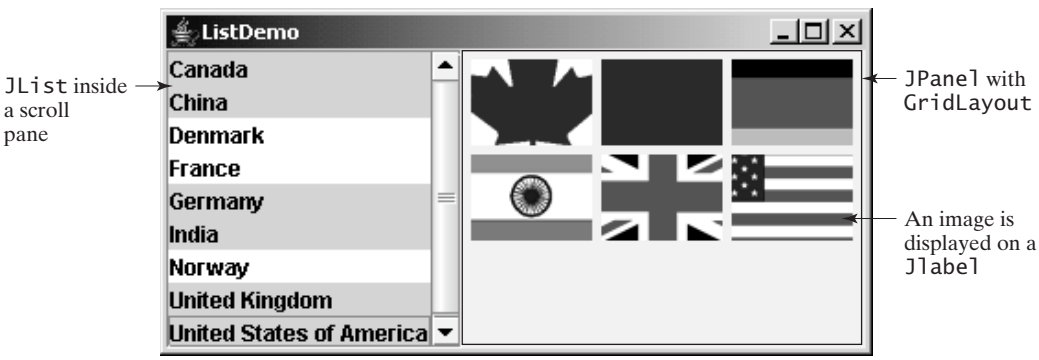


FIGURE 17.25 When the countries in the list are selected, corresponding images of their flags are displayed in the labels.

Here are the major steps in the program:

1. Create the user interface.
   Create a list with nine country names as selection values, and place the list inside a scroll pane. Place the scroll pane in the west of the frame. Create nine labels to be used to display the countries' flag images. Place the labels in the panel, and place the panel in the center of the frame.

2. Process the event.
   Create a listener to implement the `valueChanged` method in the `ListSelectionListener` interface to set the selected countries' flag images in the labels.

LISTING 17.9 ListDemo.java

```
1 import java.awt.*;
2 import javax.swing.*;
3 import javax.swing.event.*;
4
5 public class ListDemo extends JFrame {
6   final int NUMBER_OF_FLAGS = 9;
7
8   // Declare an array of Strings for flag titles
9   private String[] flagTitles = {"Canada", "China", "Denmark",
```

```
10      "France", "Germany", "India", "Norway", "United Kingdom",
11      "United States of America"};
12
13    // The list for selecting countries
14    private JList jlst = new JList(flagTitles);
15
16    // Declare an ImageIcon array for the national flags of 9 countries
17    private ImageIcon[] imageIcons = {
18      new ImageIcon("image/ca.gif"),
19      new ImageIcon("image/china.gif"),
20      new ImageIcon("image/denmark.gif"),
21      new ImageIcon("image/fr.gif"),
22      new ImageIcon("image/germany.gif"),
23      new ImageIcon("image/india.gif"),
24      new ImageIcon("image/norway.gif"),
25      new ImageIcon("image/uk.gif"),
26      new ImageIcon("image/us.gif")
27    };
28
29    // Arrays of labels for displaying images
30    private JLabel[] jlblImageViewer = new JLabel[NUMBER_OF_FLAGS];
31
32    public static void main(String[] args) {
33      ListDemo frame = new ListDemo();                                  create frame
34      frame.setSize(650, 500);
35      frame.setTitle("ListDemo");
36      frame.setLocationRelativeTo(null); // Center the frame
37      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
38      frame.setVisible(true);
39    }
40
41    public ListDemo() {
42      // Create a panel to hold nine labels
43      JPanel p = new JPanel(new GridLayout(3, 3, 5, 5));                create UI
44
45      for (int i = 0; i < NUMBER_OF_FLAGS; i++) {
46        p.add(jlblImageViewer[i] = new JLabel());
47        jlblImageViewer[i].setHorizontalAlignment
48          (SwingConstants.CENTER);
49      }
50
51      // Add p and the list to the frame
52      add(p, BorderLayout.CENTER);
53      add(new JScrollPane(jlst), BorderLayout.WEST);
54
55      // Register listeners
56      jlst.addListSelectionListener(new ListSelectionListener() {
57        /** Handle list selection */
58        public void valueChanged(ListSelectionEvent e) {                event handler
59          // Get selected indices
60          int[] indices = jlst.getSelectedIndices();
61
62          int i;
63          // Set icons in the labels
64          for (i = 0; i < indices.length; i++) {
65            jlblImageViewer[i].setIcon(imageIcons[indices[i]]);
66          }
67
68          // Remove icons from the rest of the labels
69          for (; i < NUMBER_OF_FLAGS; i++) {
```

```
70            jlblImageViewer[i].setIcon(null);
71          }
72        }
73      });
74    }
75 }
```

The anonymous inner-class listener listens to `ListSelectionEvent` for handling the selection of country names in the list (lines 56–73). `ListSelectionEvent` and `ListSelectionListener` are defined in the `javax.swing.event` package, so this package is imported in the program (line 3).

The program creates an array of nine labels for displaying flag images for nine countries. The program loads the images of the nine countries into an image array (lines 17–27) and creates a list of the nine countries in the same order as in the image array (lines 9–11). Thus the index `0` of the image array corresponds to the first country in the list.

The list is placed in a scroll pane (line 53) so that it can be scrolled when the number of items in the list extends beyond the viewing area.

By default, the selection mode of the list is multiple-interval, which allows the user to select multiple items from different blocks in the list. When the user selects countries in the list, the `valueChanged` handler (lines 58–73) is executed, which gets the indices of the selected item and sets their corresponding image icons in the label to display the flags.

17.10 Scroll Bars

`JScrollBar` is a component that enables the user to select from a range of values, as shown in Figure 17.26.

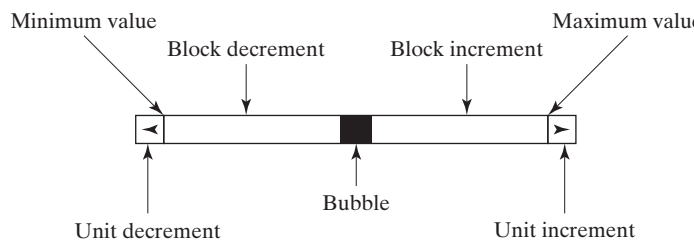


FIGURE 17.26 A scroll bar represents a range of values graphically.

Normally, the user changes the value of the scroll bar by making a gesture with the mouse. For example, the user can drag the scroll bar's bubble up and down, or click in the scroll bar's unit-increment or block-increment areas. Keyboard gestures can also be mapped to the scroll bar. By convention, the Page Up and Page Down keys are equivalent to clicking in the scroll bar's block-increment and block-decrement areas.



Note

The width of the scroll bar's track corresponds to `maximum + visibleAmount`. When a scroll bar is set to its maximum value, the left side of the bubble is at `maximum`, and the right side is at `maximum + visibleAmount`.

`JScrollBar` has the following properties, as shown in Figure 17.27.

When the user changes the value of the scroll bar, the scroll bar fires an instance of `AdjustmentEvent`, which is passed to every registered listener. An object that wishes to be notified of changes to the scroll bar's value must implement the `adjustmentValueChanged` method in the `java.awt.event.AdjustmentListener` interface.

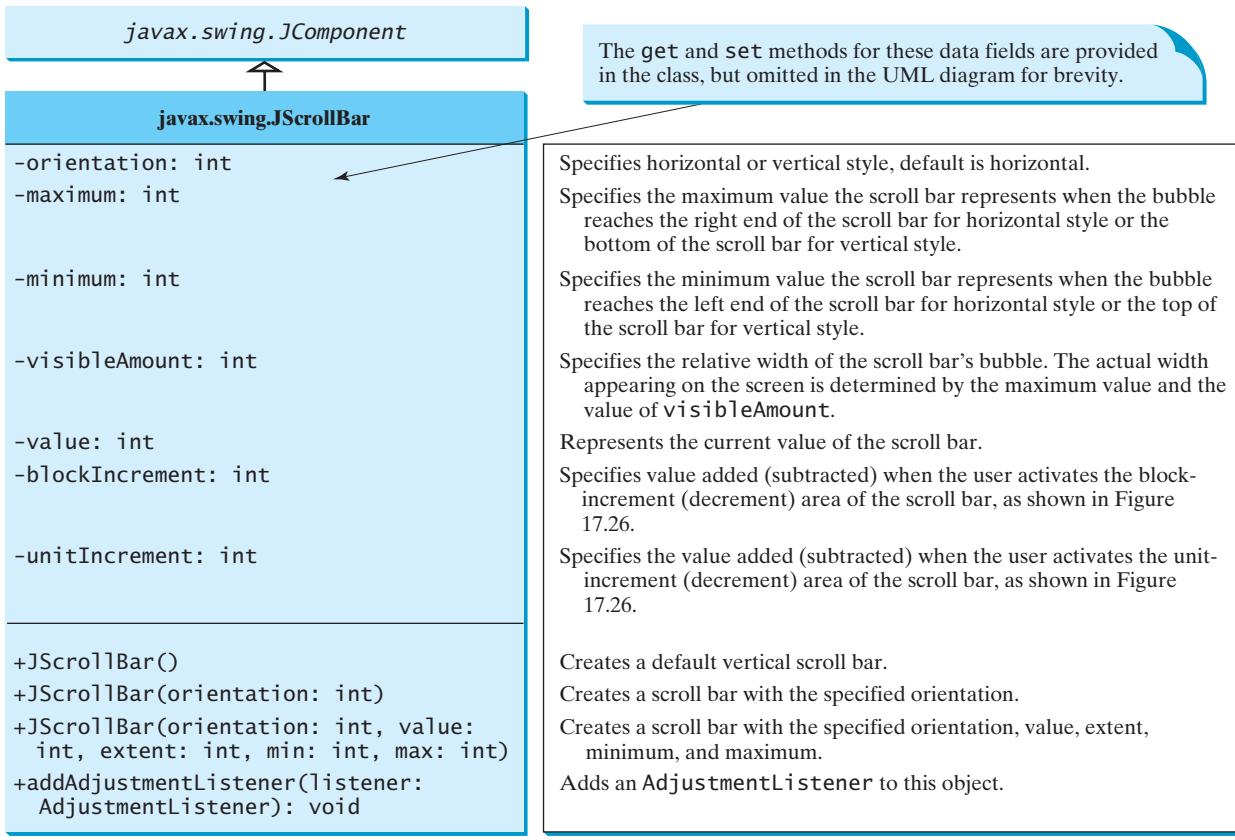

javax.swing.JComponent
javax.swing.JScrollBar
-orientation: int
-maximum: int
-minimum: int
-visibleAmount: int
-value: int
-blockIncrement: int
-unitIncrement: int
+JScrollBar()
+JScrollBar(orientation: int)
+JScrollBar(orientation: int, value: int, extent: int, min: int, max: int)
+addAdjustmentListener(listener: AdjustmentListener): void
The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
Specifies horizontal or vertical style, default is horizontal.
Specifies the maximum value the scroll bar represents when the bubble reaches the right end of the scroll bar for horizontal style or the bottom of the scroll bar for vertical style.
Specifies the minimum value the scroll bar represents when the bubble reaches the left end of the scroll bar for horizontal style or the top of the scroll bar for vertical style.
Specifies the relative width of the scroll bar's bubble. The actual width appearing on the screen is determined by the maximum value and the value of visibleAmount.
Represents the current value of the scroll bar.
Specifies value added (subtracted) when the user activates the block-increment (decrement) area of the scroll bar, as shown in Figure 17.26.
Specifies the value added (subtracted) when the user activates the unit-increment (decrement) area of the scroll bar, as shown in Figure 17.26.
Creates a default vertical scroll bar.
Creates a scroll bar with the specified orientation.
Creates a scroll bar with the specified orientation, value, extent, minimum, and maximum.
Adds an AdjustmentListener to this object.

FIGURE 17.27 `JScrollBar` enables you to select from a range of values.

Listing 17.10 gives a program that uses horizontal and vertical scroll bars to control a message displayed on a panel. The horizontal scroll bar is used to move the message to the left or the right, and the vertical scroll bar to move it up and down. A sample run of the program is shown in Figure 17.28.



FIGURE 17.28 The scroll bars move the message on a panel horizontally and vertically.

Here are the major steps in the program:

1. Create the user interface.
   Create a `MessagePanel` object and place it in the center of the frame. Create a vertical scroll bar and place it in the east of the frame. Create a horizontal scroll bar and place it in the south of the frame.

2. Process the event.
   Create listeners to implement the `adjustmentValueChanged` handler to move the message according to the bar movement in the scroll bars.

LISTING 17.10 ScrollBarDemo.java

horizontal scroll bar

vertical scroll bar

create frame

create UI

add scroll bar

adjustment listener

adjustment listener

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class ScrollBarDemo extends JFrame {
6    // Create horizontal and vertical scroll bars
7    private JScrollBar jscbHort =
8      new JScrollBar(JScrollBar.HORIZONTAL);
9    private JScrollBar jscbVert =
10     new JScrollBar(JScrollBar.VERTICAL);
11
12   // Create a MessagePanel
13   private MessagePanel messagePanel =
14     new MessagePanel("Welcome to Java");
15
16   public static void main(String[] args) {
17     ScrollBarDemo frame = new ScrollBarDemo();
18     frame.setTitle("ScrollBarDemo");
19     frame.setLocationRelativeTo(null); // Center the frame
20     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21     frame.pack();
22     frame.setVisible(true);
23   }
24
25   public ScrollBarDemo() {
26     // Add scroll bars and message panel to the frame
27     setLayout(new BorderLayout());
28     add(messagePanel, BorderLayout.CENTER);
29     add(jscbVert, BorderLayout.EAST);
30     add(jscbHort, BorderLayout.SOUTH);
31
32     // Register listener for the scroll bars
33     jscbHort.addAdjustmentListener(new AdjustmentListener() {
34       public void adjustmentValueChanged(AdjustmentEvent e) {
35         // getValue() and getMaximumValue() return int, but for better
36         // precision, use double
37         double value = jscbHort.getValue();
38         double maximumValue = jscbHort.getMaximum();
39         double newX = (value * messagePanel.getWidth() /
40           maximumValue);
41         messagePanel.setXCoordinate((int)newX);
42       }
43     });
44     jscbVert.addAdjustmentListener(new AdjustmentListener() {
45       public void adjustmentValueChanged(AdjustmentEvent e) {
46         // getValue() and getMaximumValue() return int, but for better
47         // precision, use double
48         double value = jscbVert.getValue();
49         double maximumValue = jscbVert.getMaximum();
50         double newY = (value * messagePanel.getHeight() /
51           maximumValue);
52         messagePanel.setYCoordinate((int)newY);
53       }
54     });
55   }
56 }
```

The program creates two scroll bars (`jscbVert` and `jscbHort`) (lines 7–10) and an instance of `MessagePanel` (`messagePanel`) (lines 13–14). `messagePanel` is placed in the center of the frame; `jscbVert` and `jscbHort` are placed in the east and south sections of the frame (lines 29–30), respectively.

You can specify the orientation of the scroll bar in the constructor or use the `setOrientation` method. By default, the property value is `100` for `maximum`, `0` for `minimum`, `10` for `blockIncrement`, and `10` for `visibleAmount`.

When the user drags the bubble, or clicks the increment or decrement unit, the value of the scroll bar changes. An instance of `AdjustmentEvent` is fired and passed to the listener by invoking the `adjustmentValueChanged` handler. The listener for the vertical scroll bar moves the message up and down (lines 33–43), and the listener for the horizontal bar moves the message to right and left (lines 44–54).

The maximum value of the vertical scroll bar corresponds to the height of the panel, and the maximum value of the horizontal scroll bar corresponds to the width of the panel. The ratio between the current and maximum values of the horizontal scroll bar is the same as the ratio between the `x` value and the width of the message panel. Similarly, the ratio between the current and maximum values of the vertical scroll bar is the same as the ratio between the `y` value and the height of the message panel. The `x`-coordinate and `y`-coordinate are set in response to the scroll bar adjustments (lines 39, 50).

17.11 Sliders

`JSlider` is similar to `JScrollBar`, but `JSlider` has more properties and can appear in many forms. Figure 17.29 shows two sliders.

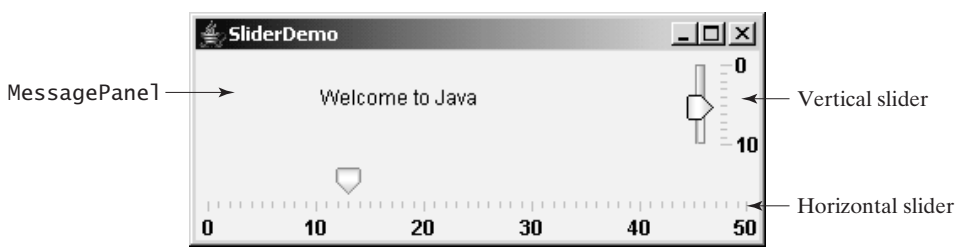


FIGURE 17.29 The sliders move the message on a panel horizontally and vertically.

`JSlider` lets the user graphically select a value by sliding a knob within a bounded interval. The slider can show both major tick marks and minor tick marks between them. The number of pixels between the tick marks is controlled by the `majorTickSpacing` and `minorTickSpacing` properties. Sliders can be displayed horizontally or vertically, with or without ticks, and with or without labels. The frequently used constructors and properties in `JSlider` are shown in Figure 17.30.



Note

The values of a vertical scroll bar increase from top to bottom, but the values of a vertical slider decrease from top to bottom by default.



Note

All the properties listed in Figure 17.30 have the associated `get` and `set` methods, which are omitted for brevity. By convention, the `get` method for a Boolean property is named *`is<PropertyName>()`*. In the `JSlider` class, the `get` methods for `paintLabels`, `paintTicks`, `paintTrack`, and `inverted` are `getPaintLabels()`, `getPaintTicks()`, `getPaintTrack()`, and `getInverted()`, which violate the naming convention.

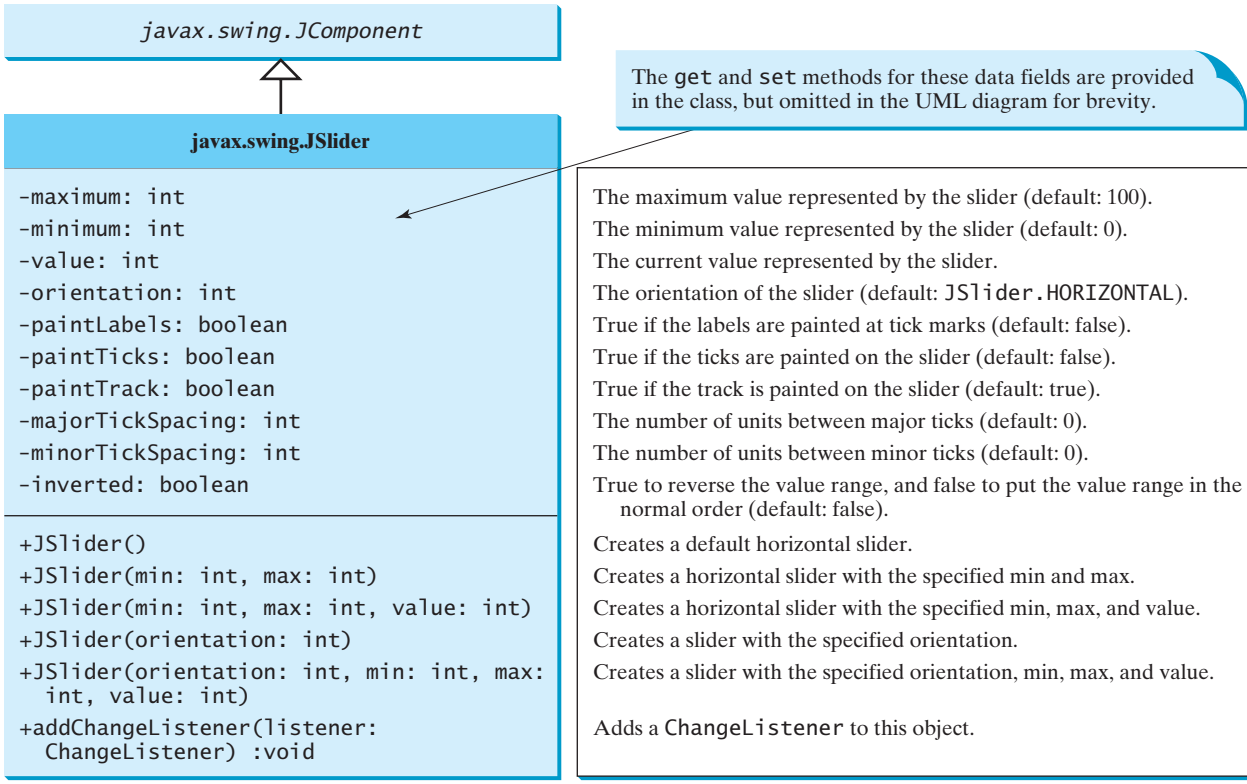

javax.swing.JComponent
javax.swing.JSlider
-maximum: int
-minimum: int
-value: int
-orientation: int
-paintLabels: boolean
-paintTicks: boolean
-paintTrack: boolean
-majorTickSpacing: int
-minorTickSpacing: int
-inverted: boolean
+JSlider()
+JSlider(min: int, max: int)
+JSlider(min: int, max: int, value: int)
+JSlider(orientation: int)
+JSlider(orientation: int, min: int, max: int, value: int)
+addChangeListener(listener: ChangeListener) :void
The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
The maximum value represented by the slider (default: 100).
The minimum value represented by the slider (default: 0).
The current value represented by the slider.
The orientation of the slider (default: JSlider.HORIZONTAL).
True if the labels are painted at tick marks (default: false).
True if the ticks are painted on the slider (default: false).
True if the track is painted on the slider (default: true).
The number of units between major ticks (default: 0).
The number of units between minor ticks (default: 0).
True to reverse the value range, and false to put the value range in the normal order (default: false).
Creates a default horizontal slider.
Creates a horizontal slider with the specified min and max.
Creates a horizontal slider with the specified min, max, and value.
Creates a slider with the specified orientation.
Creates a slider with the specified orientation, min, max, and value.
Adds a ChangeListener to this object.

FIGURE 17.30 `JSlider` enables you to select from a range of values.

When the user changes the value of the slider, the slider fires an instance of `javax.swing.event.ChangeEvent`, which is passed to any registered listeners. Any object that wishes to be notified of changes to the slider's value must implement the `stateChanged` method in the `ChangeListener` interface defined in the package `javax.swing.event`.

Listing 17.11 writes a program that uses the sliders to control a message displayed on a panel, as shown in Figure 17.29. Here are the major steps in the program:

1. Create the user interface.
   Create a `MessagePanel` object and place it in the center of the frame. Create a vertical slider and place it in the east of the frame. Create a horizontal slider and place it in the south of the frame.

2. Process the event.
   Create listeners to implement the `stateChanged` handler in the `ChangeListener` interface to move the message according to the knot movement in the slider.

LISTING 17.11 SliderDemo.java

horizontal slider
vertical slider

```
1 import java.awt.*;
2 import javax.swing.*;
3 import javax.swing.event.*;
4
5 public class SliderDemo extends JFrame {
6   // Create horizontal and vertical sliders
7   private JSlider jsldHort = new JSlider(JSlider.HORIZONTAL);
8   private JSlider jsldVert = new JSlider(JSlider.VERTICAL);
9
```

```
10   // Create a MessagePanel
11   private MessagePanel messagePanel =
12     new MessagePanel("Welcome to Java");
13
14   public static void main(String[] args) {
15     SliderDemo frame = new SliderDemo();                          create frame
16     frame.setTitle("SliderDemo");
17     frame.setLocationRelativeTo(null); // Center the frame
18     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19     frame.pack();
20     frame.setVisible(true);
21   }
22
23   public SliderDemo() {                                           create UI
24     // Add sliders and message panel to the frame
25     setLayout(new BorderLayout(5, 5));
26     add(messagePanel, BorderLayout.CENTER);
27     add(jsldVert, BorderLayout.EAST);
28     add(jsldHort, BorderLayout.SOUTH);
29
30     // Set properties for sliders
31     jsldHort.setMaximum(50);                                      slider properties
32     jsldHort.setPaintLabels(true);
33     jsldHort.setPaintTicks(true);
34     jsldHort.setMajorTickSpacing(10);
35     jsldHort.setMinorTickSpacing(1);
36     jsldHort.setPaintTrack(false);
37     jsldVert.setInverted(true);
38     jsldVert.setMaximum(10);
39     jsldVert.setPaintLabels(true);
40     jsldVert.setPaintTicks(true);
41     jsldVert.setMajorTickSpacing(10);
42     jsldVert.setMinorTickSpacing(1);
43
44     // Register listener for the sliders
45     jsldHort.addChangeListener(new ChangeListener() {             listener
46       /** Handle scroll-bar adjustment actions */
47       public void stateChanged(ChangeEvent e) {
48         // getValue() and getMaximumValue() return int, but for better
49         // precision, use double
50         double value = jsldHort.getValue();
51         double maximumValue = jsldHort.getMaximum();
52         double newX = (value * messagePanel.getWidth() /
53           maximumValue);
54         messagePanel.setXCoordinate((int)newX);
55       }
56     });
57     jsldVert.addChangeListener(new ChangeListener() {             listener
58       /** Handle scroll-bar adjustment actions */
59       public void stateChanged(ChangeEvent e) {
60         // getValue() and getMaximumValue() return int, but for better
61         // precision, use double
62         double value = jsldVert.getValue();
63         double maximumValue = jsldVert.getMaximum();
64         double newY = (value * messagePanel.getHeight() /
65           maximumValue);
66         messagePanel.setYCoordinate((int) newY);
67       }
68     });
69   }
70 }
```

`JSlider` is similar to `JScrollBar` but has more features. As shown in this example, you can specify maximum, labels, major ticks, and minor ticks on a `JSlider` (lines 31–35). You can also choose to hide the track (line 36). Since the values of a vertical slider decrease from top to bottom, the `setInverted` method reverses the order (line 37).

`JSlider` fires `ChangeEvent` when the slider is changed. The listener needs to implement the `stateChanged` handler in `ChangeListener` (lines 45–68). Note that `JScrollBar` fires `AdjustmentEvent` when the scroll bar is adjusted.

17.12 Creating Multiple Windows

Occasionally, you may want to create multiple windows in an application. The application opens a new window to perform a specified task. The new windows are called *subwindows,* and the main frame is called the *main window*.

To create a subwindow from an application, you need to define a subclass of `JFrame` that specifies the task and tells the new window what to do. You can then create an instance of this subclass in the application and launch the new window by setting the frame instance to be visible.

Listing 17.12 gives a program that creates a main window with a text area in the scroll pane and a button named *Show Histogram*. When the user clicks the button, a new window appears that displays a histogram to show the occurrences of the letters in the text area. Figure 17.31 contains a sample run of the program.

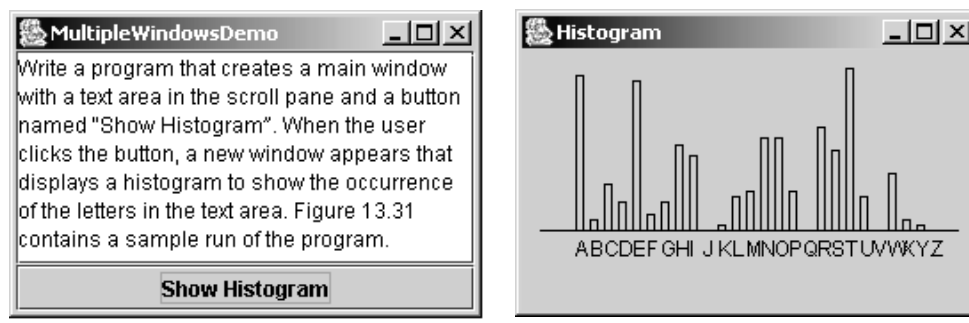

MultipleWindowsDemo
Write a program that creates a main window with a text area in the scroll pane and a button named "Show Histogram". When the user clicks the button, a new window appears that displays a histogram to show the occurrence of the letters in the text area. Figure 13.31 contains a sample run of the program.
Show Histogram
Histogram
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

FIGURE 17.31 The histogram is displayed in a separate frame.

Here are the major steps in the program:

1. Create a main class for the frame named `MultipleWindowsDemo` in Listing 17.12. Add a text area inside a scroll pane, and place the scroll pane in the center of the frame. Create a button *Show Histogram* and place it in the south of the frame.

2. Create a subclass of `JPanel` named `Histogram` in Listing 17.13. The class contains a data field named `count` of the `int[]` type, which counts the occurrences of `26` letters. The values in `count` are displayed in the histogram.

3. Implement the `actionPerformed` handler in `MultipleWindowsDemo`, as follows:

   a. Create an instance of `Histogram`. Count the letters in the text area and pass the count to the `Histogram` object.

   b. Create a new frame and place the `Histogram` object in the center of frame. Display the frame.

LISTING 17.12 MultipleWindowsDemo.java

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class MultipleWindowsDemo extends JFrame {
6    private JTextArea jta;
7    private JButton jbtShowHistogram = new JButton("Show Histogram");
8    private Histogram histogram = new Histogram();
9
10   // Create a new frame to hold the histogram panel
11   private JFrame histogramFrame = new JFrame();                    create subframe
12
13   public MultipleWindowsDemo() {                                   create UI
14     // Store text area in a scroll pane
15     JScrollPane scrollPane = new JScrollPane(jta = new JTextArea());
16     scrollPane.setPreferredSize(new Dimension(300, 200));
17     jta.setWrapStyleWord(true);
18     jta.setLineWrap(true);
19
20     // Place scroll pane and button in the frame
21     add(scrollPane, BorderLayout.CENTER);
22     add(jbtShowHistogram, BorderLayout.SOUTH);
23
24     // Register listener
25     jbtShowHistogram.addActionListener(new ActionListener() {
26       /** Handle the button action */
27       public void actionPerformed(ActionEvent e) {
28         // Count the letters in the text area
29         int[] count = countLetters();
30
31         // Set the letter count to histogram for display
32         histogram.showHistogram(count);
33
34         // Show the frame
35         histogramFrame.setVisible(true);                           display subframe
36       }
37     });
38
39     // Create a new frame to hold the histogram panel
40     histogramFrame.add(histogram);
41     histogramFrame.pack();
42     histogramFrame.setTitle("Histogram");
43   }
44
45   /** Count the letters in the text area */
46   private int[] countLetters() {
47     // Count for 26 letters
48     int[] count = new int[26];
49
50     // Get contents from the text area
51     String text = jta.getText();
52
53     // Count occurrences of each letter (case insensitive)
54     for (int i = 0; i < text.length(); i++) {
55       char character = text.charAt(i);
56
```

```
57        if ((character >= 'A') && (character <= 'Z')) {
58          count[character - 'A']++;
59        }
60        else if ((character >= 'a') && (character <= 'z')) {
61          count[character - 'a']++;
62        }
63      }
64
65      return count; // Return the count array
66    }
67
68    public static void main(String[] args) {
69      MultipleWindowsDemo frame = new MultipleWindowsDemo();
70      frame.setLocationRelativeTo(null); // Center the frame
71      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
72      frame.setTitle("MultipleWindowsDemo");
73      frame.pack();
74      frame.setVisible(true);
75    }
76  }
```

create main frame

LISTING 17.13 Histogram.java

```
1  import javax.swing.*;
2  import java.awt.*;
3
4  public class Histogram extends JPanel {
5    // Count the occurrences of 26 letters
6    private int[] count;
7
8    /** Set the count and display histogram */
9    public void showHistogram(int[] count) {
10     this.count = count;
11     repaint();
12   }
13
14   /** Paint the histogram */
15   protected void paintComponent(Graphics g) {
16     if (count == null) return; // No display if count is null
17
18     super.paintComponent(g);
19
20     // Find the panel size and bar width and interval dynamically
21     int width = getWidth();
22     int height = getHeight();
23     int interval = (width - 40) / count.length;
24     int individualWidth = (int)(((width - 40) / 24) * 0.60);
25
26     // Find the maximum count. The maximum count has the highest bar
27     int maxCount = 0;
28     for (int i = 0; i < count.length; i++) {
29       if (maxCount < count[i])
30         maxCount = count[i];
31     }
32
33     // x is the start position for the first bar in the histogram
34     int x = 30;
35
36     // Draw a horizontal base line
```

paint histogram

```
37      g.drawLine(10, height - 45, width - 10, height - 45);
38      for (int i = 0; i < count.length; i++) {
39        // Find the bar height
40        int barHeight =
41          (int)(((double)count[i] / (double)maxCount) * (height - 55));
42
43        // Display a bar (i.e. rectangle)
44        g.drawRect(x, height - 45 - barHeight, individualWidth,
45          barHeight);
46
47        // Display a letter under the base line
48        g.drawString((char)(65 + i) + "", x, height - 30);
49
50        // Move x for displaying the next character
51        x += interval;
52      }
53    }
54
55    /** Override getPreferredSize */
56    public Dimension getPreferredSize() {                    preferredSize
57      return new Dimension(300, 300);
58    }
59  }
```

The program contains two classes: `MultipleWindowsDemo` and `Histogram`. Their relationship is shown in Figure 17.32.

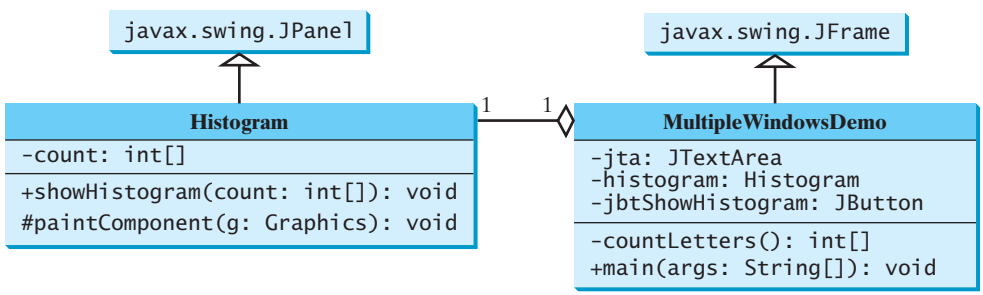


FIGURE 17.32 `MultipleWindowsDemo` uses `Histogram` to display a histogram of the occurrences of the letters in a text area in the frame.

`MultipleWindowsDemo` is a frame that holds a text area in a scroll pane and a button. `Histogram` is a subclass of `JPanel` that displays a histogram for the occurrences of letters in the text area.

When the user clicks the *Show Histogram* button, the handler counts the occurrences of letters in the text area. Letters are counted regardless of their case. Nonletter characters are not counted. The count is stored in an `int` array of `26` elements. The first element in the array stores the count for letter `'a'` or `'A'`, and the last element stores the count for letter `'z'` or `'Z'`. The `count` array is passed to the histogram for display.

The `MultipleWindowsDemo` class contains a `main` method. The `main` method creates an instance of `MultipleWindowsDemo` and displays the frame. The `MultipleWindowsDemo` class also contains an instance of `JFrame`, named `histogramFrame`, which holds an instance of `Histogram`. When the user clicks the *Show Histogram* button, `histogramFrame` is set visible to display the histogram.

The height and width of the bars in the histogram are determined dynamically according to the window size of the histogram.

You cannot add an instance of `JFrame` to a container. For example, adding `histogramFrame` to the main frame would cause a runtime exception. However, you can create a frame instance and set it visible to launch a new window.

Chapter Summary

1. You learned how to create graphical user interfaces using Swing GUI components `JButton`, `JCheckBox`, `JRadioButton`, `JLabel`, `JTextField`, `JTextArea`, `JComboBox`, `JList`, `JScrollBar`, and `JSlider`. You also learned how to handle events on these components.
2. You can display a text and icon on buttons (`JButton`, `JCheckBox`, `JRadioButton`) and labels (`JLabel`).

Review Questions

Sections 17.2–17.4

17.1 How do you create a button labeled OK? How do you change text on a button? How do you set an icon, a pressed icon, and a rollover icon in a button?

17.2 Given a `JButton` object `jbtOK`, write statements to set the button's foreground to `red`, background to `yellow`, mnemonic to `'K'`, tool tip text to `"Click OK to proceed"`, horizontal alignment to `RIGHT`, vertical alignment to `BOTTOM`, horizontal text position to `LEFT`, vertical text position to `TOP`, and icon text gap to `5`.

17.3 How do you create a check box? How do you create a check box with the box checked initially? How do you determine whether a check box is selected?

17.4 How do you create a radio button? How do you create a radio button with the button selected initially? How do you group the radio buttons together? How do you determine whether a radio button is selected?

17.5 List at least five properties defined in the `AbstractButton` class.

Sections 17.5–17.9

17.6 How do you create a label named `"Address"`? How do you change the name on a label? How do you set an icon in a label?

17.7 Given a `JLabel` object `jlblMap`, write statements to set the label's foreground to `red`, background to `yellow`, mnemonic to `'K'`, tool tip text to `"Map image"`, horizontal alignment to `RIGHT`, vertical alignment to `BOTTOM`, horizontal text position to `LEFT`, vertical text position to `TOP`, and icon text gap to `5`.

17.8 How do you create a text field with `10` columns and the default text `"Welcome to Java"`? How do you write the code to check whether a text field is empty?

17.9 How do you create a text area with ten rows and 20 columns? How do you insert three lines into the text area? How do you create a scrollable text area?

17.10 How do you create a combo box, add three items to it, and retrieve a selected item?

17.11 How do you create a list with an array of strings?

Sections 17.10–17.12

17.12 How do you create a horizontal scroll bar? What event can a scroll bar fire?

17.13 How do you create a vertical slider? What event can a slider fire?

17.14 Explain how to create and show multiple frames in an application.

PROGRAMMING EXERCISES



Pedagogical Note

Instructors may assign the exercises from the next chapter as exercises for this chapter. Instead of writing Java applets, ask students to write Java applications.

additional exercises

Sections 17.2–17.5

17.1* (*Revising Listing 17.2, ButtonDemo.java*) Rewrite Listing 17.2 to add a group of radio buttons to select background colors. The available colors are red, yellow, white, gray, and green (see Figure 17.33).



FIGURE 17.33 The <= and => buttons move the message on the panel, and you can also set the background color for the message.

17.2* (*Selecting geometric figures*) Write a program that draws various figures, as shown in Figure 17.34. The user selects a figure from a radio button and specifies whether it is filled in a check box. (*Hint*: Use the `FigurePanel` class introduced in Listing 15.3 to display a figure.)

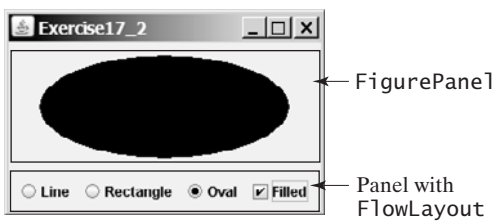


FIGURE 17.34 The program displays lines, rectangles, and ovals when you select a shape type.

17.3** (*Traffic lights*) Write a program that simulates a traffic light. The program lets the user select one of three lights: red, yellow, or green. When a radio button is selected, the light is turned on, and only one light can be on at a time (see Figure 17.35). No light is on when the program starts.

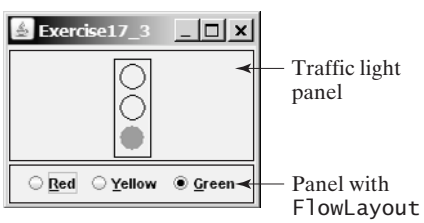
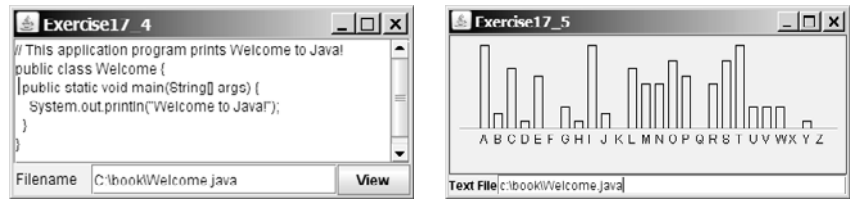


FIGURE 17.35 The radio buttons are grouped to let you select only one color in the group to control a traffic light.

Sections 17.6–17.10

17.4** (*Text Viewer*) Write a program that displays a text file in a text area, as shown in Figure 17.36. The user enters a file name in a text field and clicks the *View* button; the file is then displayed in a text area.


Exercise17_4
// This application program prints Welcome to Java!
public class Welcome {
public static void main(String[] args) {
System.out.println("Welcome to Java!");
}
}
Filename
C:\book\Welcome.java
View
Exercise17_5
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Text File
c:\book\Welcome.java

(a) (b)

FIGURE 17.36 (a) The program displays the text from a file to a text area. (b) The program displays a histogram that shows the occurrences of each letter in the file.

17.5** (*Creating a histogram for occurrences of letters*) In Listing 17.12, MultipleWindowsDemo.java, you developed a program that displays a histogram to show the occurrences of each letter in a text area. Reuse the `Histogram` class created in Listing 17.12 to write a program that will display a histogram on a panel. The histogram should show the occurrences of each letter in a text file, as shown in Figure 17.36(b). Assume that the letters are not case sensitive.

- Place a panel that will display the histogram in the center of the frame.
- Place a label and a text field in a panel, and put the panel in the south side of the frame. The text file will be entered from this text field.
- Pressing the *Enter* key on the text field causes the program to count the occurrences of each letter and display the count in a histogram.

17.6* (*Creating a miles/kilometers converter*) Write a program that converts miles and kilometers, as shown in Figure 17.37. If you enter a value in the Mile text field and press the *Enter* key, the corresponding kilometer is displayed in the Kilometer text field. Likewise, if you enter a value in the Kilometer text field and press the *Enter* key, the corresponding mile is displayed in the Mile text field.

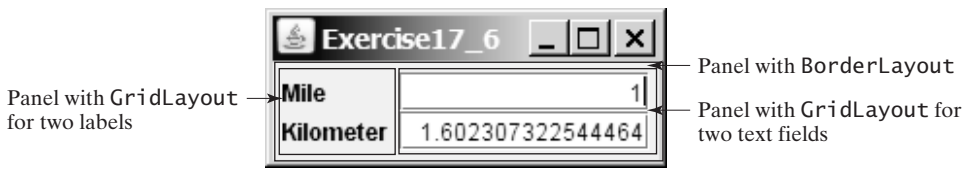


FIGURE 17.37 The program converts miles to kilometers, and vice versa.

17.7* (*Setting clock time*) Write a program that displays a clock time and sets it with the input from three text fields, as shown in Figure 17.38. Use the `StillClock` in Listing 15.10.

17.8** (*Selecting a font*) Write a program that can dynamically change the font of a message to be displayed on a panel. The message can be displayed in bold and italic at the same time, or can be displayed in the center of the panel. You can select the font name or font size from combo boxes, as shown in Figure 17.39. The available font names can be obtained using `getAvailableFontFamilyNames()` in `GraphicsEnvironment` (§12.8, "The `Font` Class"). The combo box for font size is initialized with numbers from `1` to `100`.

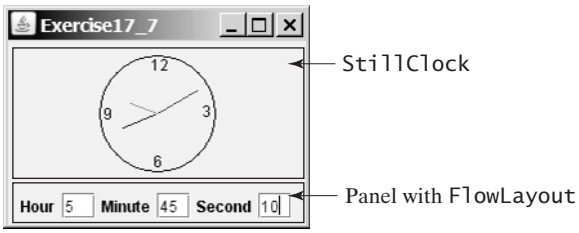


FIGURE 17.38 The program displays the time specified in the text fields.



FIGURE 17.39 You can dynamically set the font for the message.

17.9** (*Demonstrating `JLabel` properties*) Write a program to let the user dynamically set the properties `horizontalAlignment`, `verticalAlignment`, `horizontalTextAlignment`, and `verticalTextAlignment`, as shown in Figure 17.40.

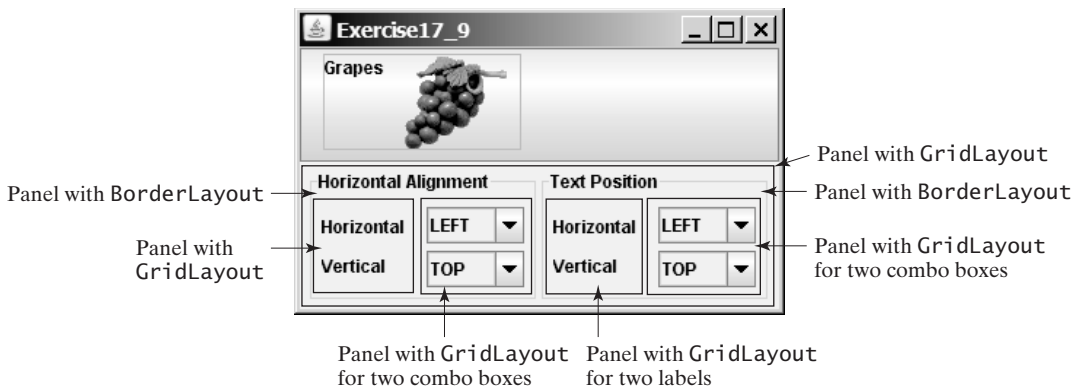


FIGURE 17.40 You can set the alignment and text-position properties of a button dynamically.

17.10* (*Adding new features into Listing 17.2, ButtonDemo.java, incrementally*) Improve Listing 17.2 incrementally, as follows (see Figure 17.41):

1. Add a text field labeled `"Enter a new message"` in the same panel with the buttons. When you type a new message in the text field and press the *Enter* key, the new message is displayed in the message panel.
2. Add a combo box labeled `"Select an interval"` in the same panel with the buttons. The combo box enables the user to select a new interval for moving the message. The selection values range from `5` to `100` with interval `5`. The user can also type a new interval in the combo box.
3. Add three radio buttons that enable the user to select the foreground color for the message as `Red`, `Green`, and `Blue`. The radio buttons are grouped in a panel, and the panel is placed in the north of the frame's content pane.

4. Add three check boxes that enable the user to center the message and display it in italic or bold. Place the check boxes in the same panel with the radio buttons.
5. Add a border titled `"Message Panel"` on the message panel, add a border titled `"South Panel"` on the panel for buttons, and add a border titled `"North Panel"` on the panel for radio buttons and check boxes.

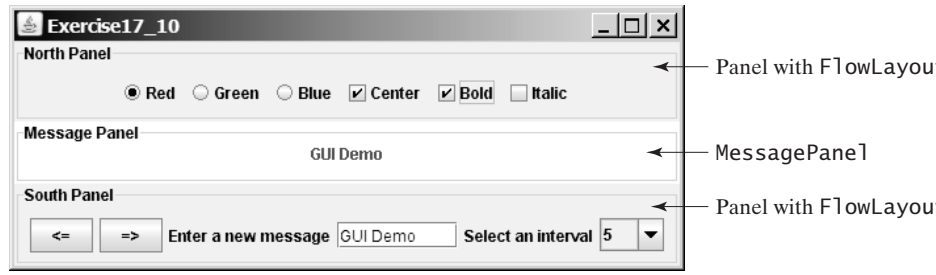


FIGURE 17.41 The program uses buttons, labels, text fields, combo boxes, radio buttons, check boxes, and borders.

17.11* (*Demonstrating `JTextField` properties*) Write a program that sets the horizontal-alignment and column-size properties of a text field dynamically, as shown in Figure 17.42.

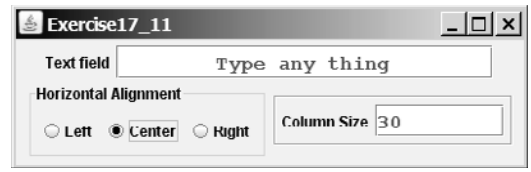


FIGURE 17.42 You can set the horizontal-alignment and column-size properties of a text field dynamically.



Video Note
Use text areas

17.12* (*Demonstrating `JTextArea` properties*) Write a program that demonstrates the wrapping styles of the text area. The program uses a check box to indicate whether the text area is wrapped. In the case where the text area is wrapped, you need to specify whether it is wrapped by characters or by words, as shown in Figure 17.43.

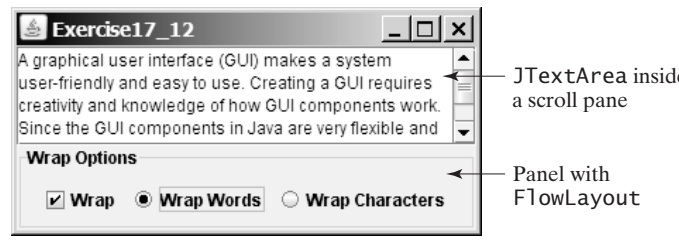


FIGURE 17.43 You can set the options to wrap a text area by characters or by words dynamically.

17.13* (*Comparing loans with various interest rates*) Rewrite Exercise 4.21 to create a user interface, as shown in Figure 17.44. Your program should let the user enter the loan amount and loan period in number of years from a text field, and should display the monthly and total payments for each interest rate starting from 5 percent to 8 percent, with increments of one-eighth, in a text area.

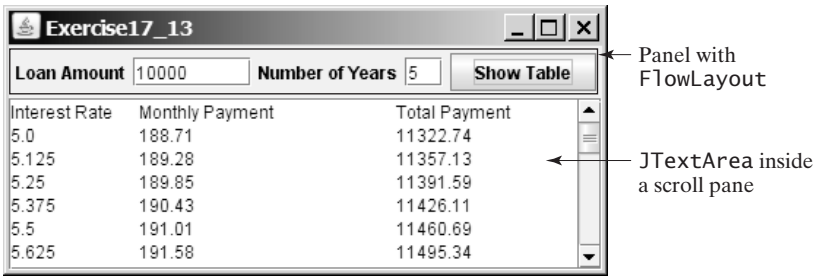


FIGURE 17.44 The program displays a table for monthly payments and total payments on a given loan based on various interest rates.

17.14* (*Using `JComboBox` and `JList`*) Write a program that demonstrates selecting items in a list. The program uses a combo box to specify a selection mode, as shown in Figure 17.45. When you select items, they are displayed in a label below the list.

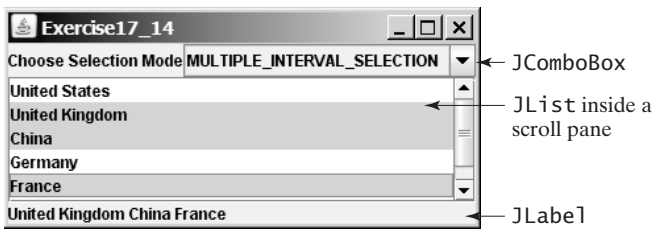


FIGURE 17.45 You can choose single selection, single-interval selection, or multiple-interval selection in a list.

Sections 17.11–17.13

17.15** (*Using `JScrollBar`*) Write a program that uses scroll bars to select the foreground color for a label, as shown in Figure 17.46. Three horizontal scroll bars are used for selecting the red, green, and blue components of the color. Use a title border on the panel that holds the scroll bars.

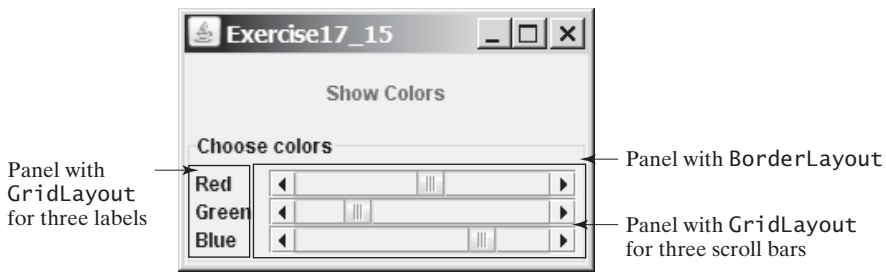


FIGURE 17.46 The foreground color changes in the label as you adjust the scroll bars.

17.16** (*Using `JSlider`*) Revise the preceding exercise using sliders.

17.17*** (*Displaying a calendar*) Write a program that displays the calendar for the current month, as shown in Figure 17.47. Use labels, and set texts on the labels to display the calendar. Use the `GregorianCalendar` class in §14.3, "Example: `Calendar` and `GregorianCalendar`," to obtain the information about month, year, first day of the month, and number of days in the month.

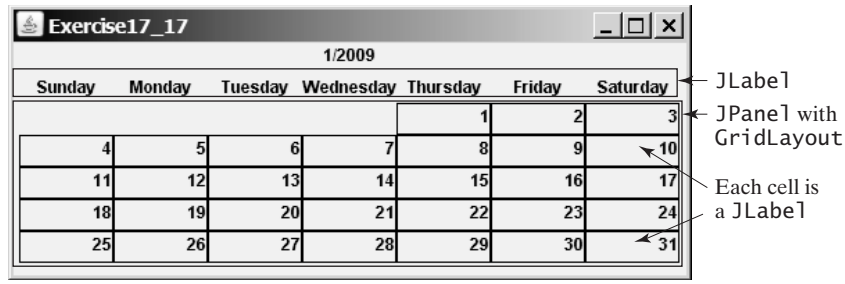


FIGURE 17.47 The program displays the calendar for the current month.

17.18* (*Revising Listing 17.12, MultipleWindowsDemo.java*) Instead of displaying the occurrences of the letters using the `Histogram` component in Listing 17.12, use a bar chart, so that the display is as shown in Figure 17.48.

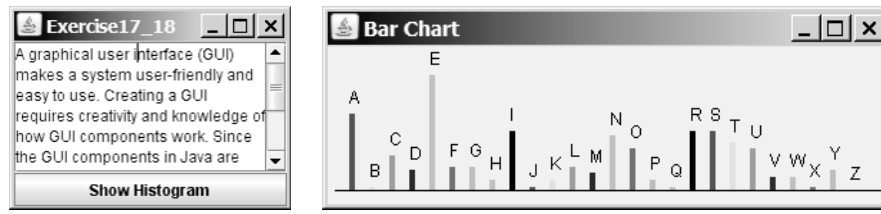

Exercise17_18
A graphical user interface (GUI) makes a system user-friendly and easy to use. Creating a GUI requires creativity and knowledge of how GUI components work. Since the GUI components in Java are
Show Histogram
Bar Chart
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

FIGURE 17.48 The number of occurrences of each letter is displayed in a bar chart.

17.19** (*Displaying country flag and flag description*) Listing 17.8, ComboBoxDemo.java, gives a program that lets users view a country's flag image and description by selecting the country from a combo box. The description is a string coded in the program. Rewrite the program to read the text description from a file. Suppose that the descriptions are stored in the file **description0.txt**, . . . , and **description8.txt** under the `text` directory for the nine countries Canada, China, Denmark, France, Germany, India, Norway, the United Kingdom, and the United States, in this order.

17.20** (*Slide show*) Exercise 16.13 developed a slide show using images. Rewrite Exercise 16.13 to develop a slide show using text files. Suppose ten text files named `slide0.txt`, `slide1.txt`, . . . , and `slide9.txt` are stored in the `text` directory. Each slide displays the text from one file. Each slide is shown for a second. The slides are displayed in order. When the last slide finishes, the first slide is redisplayed, and so on. Use a text area to display the slide.

CHAPTER 18

APPLETS AND MULTIMEDIA

Objectives

- To convert GUI applications to applets (§18.2).
- To embed applets in Web pages (§18.3).
- To run applets from Web browsers and from the appletviewer (§§18.3.1–18.3.2).
- To understand the applet security sandbox model (§18.4).
- To write a Java program that can run both as an application and as an applet (§18.5).
- To override the applet life-cycle methods `init`, `start`, `stop`, and `destroy` (§18.6).
- To pass string values to applets from HTML (§18.7).
- To develop an animation for a bouncing ball (§18.8).
- To develop an applet for the TicTacToe game (§18.9).
- To locate resources (images and audio) using the `URL` class (§18.10).
- To play audio in any Java program (§18.11).



18.1 Introduction

When browsing the Web, often the graphical user interface and animation you see are developed by the use of Java. The programs used are called Java applets. Suppose you want to develop a Java applet for the Sudoku game, as shown in Figure 18.1. How do you write this program?

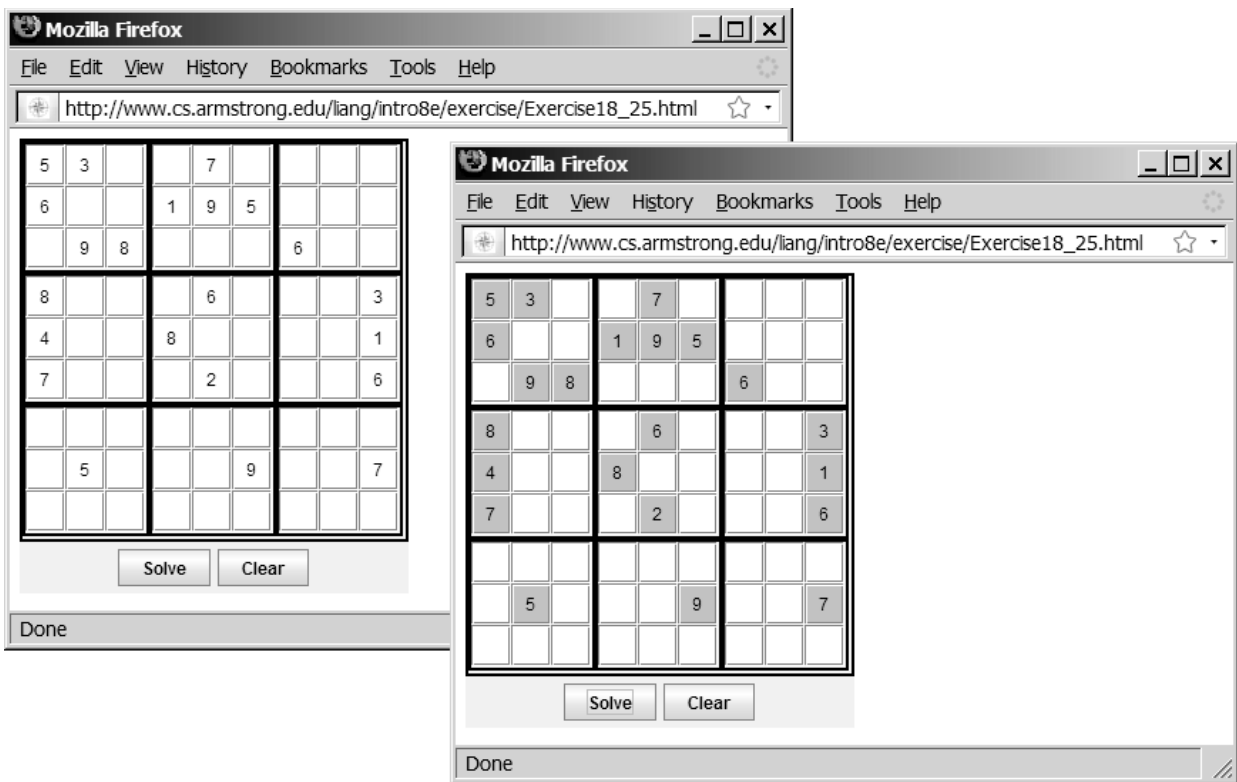


FIGURE 18.1 The Sudoku game is displayed in a Web browser.

In this chapter, you will learn how to write Java applets, explore the relationship between applets and the Web browser, and discover the similarities and differences between applications and applets. You will also learn how to create multimedia Java applications and applets with images and audio.

18.2 Developing Applets



Video Note
First applet

So far, you have used only Java applications. Everything you have learned about writing applications, however, applies also to writing applets. Applications and applets share many common programming features, although they differ slightly in some aspects. For example, every application must have a `main` method, which is invoked by the Java interpreter. Java applets, on the other hand, do not need a `main` method. They run in the Web browser environment. Because applets are invoked from a Web page, Java provides special features that enable applets to run from a Web browser.

The `Applet` class provides the essential framework that enables applets to be run from a Web browser. While every Java application has a `main` method that is executed when the application starts, applets do not have a `main` method. Instead they depend on the browser to run. Every applet is a subclass of `java.applet.Applet`. The `Applet` class is an AWT class

and is not designed to work with Swing components. To use Swing components in Java applets, you need to create a Java applet that extends `javax.swing.JApplet`, which is a subclass of `java.applet.Applet`.

Every Java GUI program you have developed can be converted into an applet by replacing `JFrame` with `JApplet` and deleting the `main` method. Figure 18.2(a) shows a Java GUI application program, which can be converted into a Java applet as shown in Figure 18.2(b).

```
import javax.swing.*;

public class DisplayLabel extends JFrame {
  public DisplayLabel() {
    add(new JLabel("Great!", JLabel.CENTER));
  }

  public static void main(String[] args) {
    JFrame frame = new DisplayLabel();
    frame.setTitle("DisplayLabel");
    frame.setSize(200, 100);
    frame.setLocationRelativeTo(null);
    frame.setDefaultCloseOperation(
      JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

(a) GUI application

```
import javax.swing.*;
                                        JApplet
public class DisplayLabel extends JFrame {
  public DisplayLabel() {
    add(new JLabel("Great!", JLabel.CENTER));
  }

  public static void main(String[] args) {
    JFrame frame = new DisplayLabel();
    frame.setTitle("DisplayLabel");
    frame.setSize(200, 100);
    frame.setLocationRelativeTo(null);
    frame.setDefaultCloseOperation(
      JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

(b) Applet

FIGURE 18.2 You can convert a GUI application into an applet.

Listing 18.1 gives the complete code for the applet.

LISTING 18.1 DisplayLabel.java

```
1 import javax.swing.*;
2
3 public class DisplayLabel extends JApplet {
4   public DisplayLabel() {
5     add(new JLabel("Great!", JLabel.CENTER));
6   }
7 }
```

extend **JApplet**

Like `JFrame`, `JApplet` is a container that can contain other GUI components (see the GUI class diagrams in Figure 14.1).

18.3 The HTML File and the <applet> Tag

To run an applet from a browser, you need to create an HTML file with the `<applet>` tag.

HTML is a markup language that presents static documents on the Web. It uses tags to instruct the Web browser how to render a Web page and contains a tag called `<applet>` that incorporates applets into a Web page.

The HTML file in Listing 18.2 invokes the **DisplayLabel.class**:

LISTING 18.2 DisplayLabel.html

```
<html>
  <head>
    <title>Java Applet Demo</title>
  </head>
  <body>
```

applet class

```
    <applet
      code = "DisplayLabel.class"
      width = 250
      height = 50>
    </applet>
  </body>
</html>
```

A *tag* is an instruction to the Web browser. The browser interprets the tag and decides how to display or otherwise treat the subsequent contents of the HTML document. Tags are enclosed inside brackets. The first word in a tag, called the *tag name*, describes tag functions. A tag can have additional attributes, sometimes with values after an equals sign, which further define the tag's action. For example, in the preceding HTML file, `<applet>` is the tag name, and `code`, `width`, and `height` are attributes. The `width` and `height` attributes specify the rectangular viewing area of the applet.

Most tags have a *start tag* and a corresponding *end tag*. The tag has a specific effect on the region between the start tag and the end tag. For example, `<applet...>...</applet>` tells the browser to display an applet. An end tag is always the start tag's name preceded by a slash.

HTML tag

An HTML document begins with the `<html>` tag, which declares that the document is written in HTML. Each document has two parts, a *head* and a *body*, defined by `<head>` and `<body>` tags, respectively. The head part contains the document title, including the `<title>` tag and other information the browser can use when rendering the document, and the body part holds the actual contents of the document. The header is optional. For more information, refer to Supplement V.A, "HTML and XHTML Tutorial."

<applet> tag

The complete syntax of the `<applet>` tag is as follows:

```
<applet
  [codebase = applet_url]
  code = classfilename.class
  width = applet_viewing_width_in_pixels
  height = applet_viewing_height_in_pixels
  [archive = archivefile]
  [vspace = vertical_margin]
  [hspace = horizontal_margin]
  [align = applet_alignment]
  [alt = alternative_text]
>
<param name = param_name1 value = param_value1>
<param name = param_name2 value = param_value2>
...
<param name = param_name3 value = param_value3>
</applet>
```

<param> tag

The `code`, `width`, and `height` attributes are required; all the others are optional. The `<param>` tag will be introduced in §18.7, "Passing Strings to Applets." The other attributes are explained below.

codebase attribute

- **`codebase`** specifies a base where your classes are loaded. If this attribute is not used, the Web browser loads the applet from the directory in which the HTML page is located. If your applet is located in a different directory from the HTML page, you must specify the `applet_url` for the browser to load the applet. This attribute enables you to load the class from anywhere on the Internet. The classes used by the applet are dynamically loaded when needed.

archive attribute

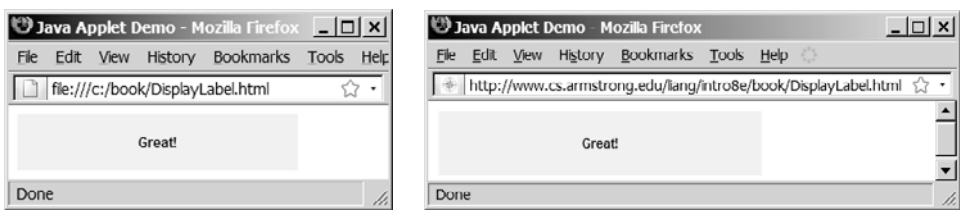
- **`archive`** instructs the browser to load an archive file that contains all the class files needed to run the applet. Archiving allows the Web browser to load all the classes

from a single compressed file at one time, thus reducing loading time and improving performance. To create archives, see Supplement III.Q, "Packaging and Deploying Java Projects."

- **vspace** and **hspace** specify the size, in pixels, of the blank margin to pad around the applet vertically and horizontally.
- **align** specifies how the applet will be aligned in the browser. One of nine values is used: **left**, **right**, **top**, **texttop**, **middle**, **absmiddle**, **baseline**, **bottom**, or **absbottom**.
- **alt** specifies the text to be displayed in case the browser cannot run Java.

18.3.1 Viewing Applets from a Web Browser

To display an applet from a Web browser, open the applet's HTML file (e.g., DisplayLabel.html). Its output is shown in Figure 18.3(a).



(a) (b)

FIGURE 18.3 The DisplayLabel program is loaded from a local host in (a) and from a Web server in (b).

To make your applet accessible on the Web, you need to store the DisplayLabel.class and DisplayLabel.html on a Web server, as shown in Figure 18.4. You can view the applet from an appropriate URL. For example, I have uploaded these two files on Web server www.cs.armstrong.edu/. As shown in Figure 18.3(b), you can access the applet from www.cs.armstrong.edu/liang/intro8e/book/DisplayLabel.html.

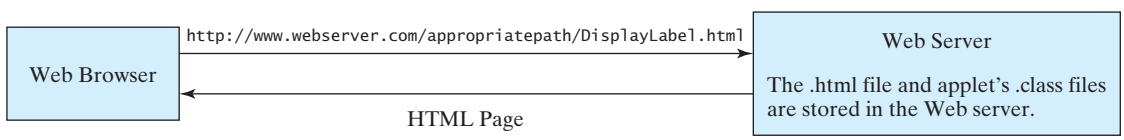

Web Browser
http://www.webserver.com/appropriatepath/DisplayLabel.html
HTML Page
Web Server
The .html file and applet's .class files are stored in the Web server.

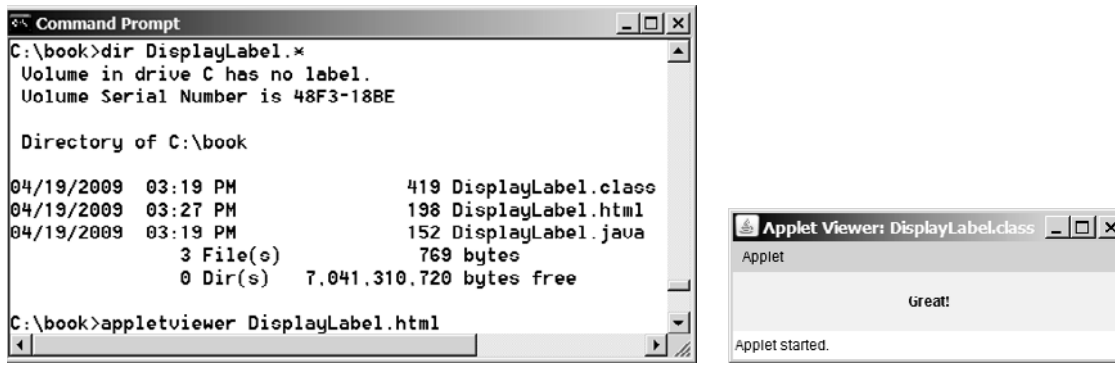
FIGURE 18.4 A Web browser requests an HTML page from a Web server.

18.3.2 Viewing Applets Using the Applet Viewer Utility

You can test the applet using the applet viewer utility, which can be invoked from the DOS prompt using the **appletviewer** command from **c:\book**, as shown in Figure 18.5(a). Its output is shown in Figure 18.5(b).

appletviewer

The applet viewer functions as a browser. It is convenient for testing applets during development without launching a Web browser.



(a) (b)

FIGURE 18.5 The appletviewer command runs a Java applet in the applet viewer utility.

18.4 Applet Security Restrictions

Java uses the so-called "sandbox security model" for executing applets to prevent destructive programs from damaging the system on which the browser is running. Applets are not allowed to use resources outside the "sandbox." Specifically, the sandbox restricts the following activities:

- Applets are not allowed to read from, or write to, the file system of the computer. Otherwise, they could damage the files and spread viruses.
- Applets are not allowed to run programs on the browser's computer. Otherwise, they might call destructive local programs and damage the local system on the user's computer.
- Applets are not allowed to establish connections between the user's computer and any other computer, except for the server where the applets are stored. This restriction prevents the applet from connecting the user's computer to another computer without the user's knowledge.

signed applet



Note

You can create *signed applets* to circumvent the security restrictions. See java.sun.com/developer/onlineTraining/Programming/JDCBook/signed.html for detailed instructions on how to create signed applets.

18.5 Enabling Applets to Run as Applications



Video Note
Run applets standalone

Despite some differences, the `JFrame` class and the `JApplet` class have a lot in common. Since they both are subclasses of the `Container` class, all their user-interface components, layout managers, and event-handling features are the same. Applications, however, are invoked from the static `main` method by the Java interpreter, and applets are run by the Web browser. The Web browser creates an instance of the applet using the applet's no-arg constructor and controls and executes the applet.

In general, an applet can be converted to an application without loss of functionality. An application can be converted to an applet as long as it does not violate the security restrictions imposed on applets. You can implement a `main` method in an applet to enable the applet to run as an application. This feature has both theoretical and practical implications. Theoretically, it blurs the difference between applets and applications. You can write a class that is both an applet and an application. From the standpoint of practicality, it is convenient to be able to run a program in two ways.

How do you write such program? Suppose you have an applet named **MyApplet**. To enable it to run as an application, you need only add a **main** method in the applet with the implementation, as follows:

```
public static void main(String[] args) {
  // Create a frame
  JFrame frame = new JFrame("Applet is in the frame");

  // Create an instance of the applet
  MyApplet applet = new MyApplet();

  // Add the applet to the frame
  frame.add(applet, BorderLayout.CENTER);

  // Display the frame
  frame.setSize(300, 300);
  frame.setLocationRelativeTo(null); // Center the frame
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setVisible(true);
}
```

create frame

create applet

add applet

show frame

You can revise the **DisplayLabel** class in Listing 18.1 to enable it to run standalone by adding a **main** method in Listing 18.3.

LISTING 18.3 New DisplayLabel.java with a main Method

```
1  import javax.swing.*;
2
3  public class DisplayLabel extends JApplet {
4    public DisplayLabel() {
5      add(new JLabel("Great!", JLabel.CENTER));
6    }
7
8    public static void main(String[] args) {
9      // Create a frame
10     JFrame frame = new JFrame("Applet is in the frame");
11
12     // Create an instance of the applet
13     DisplayLabel applet = new DisplayLabel();
14
15     // Add the applet to the frame
16     frame.add(applet);
17
18     // Display the frame
19     frame.setSize(300, 100);
20     frame.setLocationRelativeTo(null); // Center the frame
21     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22     frame.setVisible(true);
23   }
24 }
```

new main method

When the applet is run from a Web browser, the browser creates an instance of the applet and displays it. When the applet is run standalone, the main method is invoked to create a frame (line 10) to hold the applet. The applet is created (line 13) and added to the frame (line 16). The frame is displayed in line 22.

18.6 Applet Life-Cycle Methods

applet container

Applets are actually run from the *applet container*, which is a plug-in of a Web browser. The `Applet` class contains the `init()`, `start()`, `stop()`, and `destroy()` methods, known as the *life-cycle methods*. These methods are called by the applet container to control the execution of an applet. They are implemented with an empty body in the `Applet` class. So, they do nothing by default. You may override them in a subclass of `Applet` to perform desired operations. Figure 18.6 shows how the applet container calls these methods.

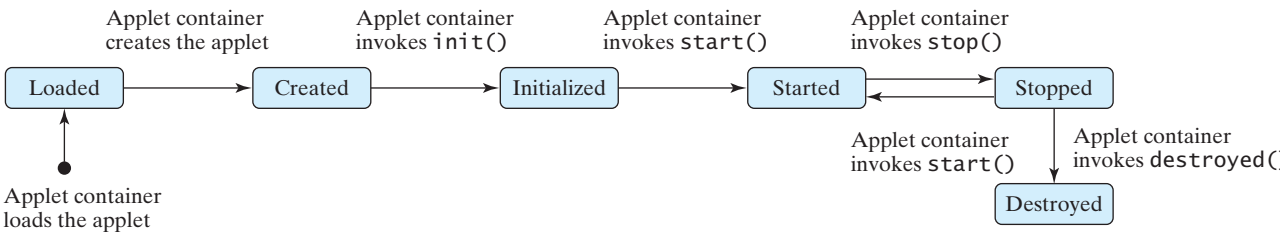


FIGURE 18.6 The applet container uses the `init`, `start`, `stop`, and `destroy` methods to control the applet.

18.6.1 The `init` Method

`init()`

The `init` method is invoked after the applet is created. If a subclass of `Applet` has an initialization to perform, it should override this method. The functions usually implemented in this method include getting string parameter values from the `<applet>` tag in the HTML page.

18.6.2 The `start` Method

`start()`

The `start` method is invoked after the `init` method. It is also called when the user returns to the Web page containing the applet after surfing other pages.

A subclass of `Applet` overrides this method if it has any operation that needs to be performed whenever the Web page containing the applet is visited. An applet with animation, for example, might start the timer to resume animation.

18.6.3 The `stop` Method

`stop()`

The `stop` method is the opposite of the `start` method. The `start` method is called when the user moves back to the page that contains the applet. The `stop` method is invoked when the user leaves the page.

A subclass of `Applet` overrides this method if it has any operation to be performed each time the Web page containing the applet is no longer visible. An applet with animation, for example, might stop the timer to pause animation.

18.6.4 The `destroy` Method

`destroy()`

The `destroy` method is invoked when the browser exits normally to inform the applet that it is no longer needed and should release any resources it has allocated. The `stop` method is always called before the `destroy` method.

A subclass of `Applet` overrides this method if it has any operation to be performed before it is destroyed. Usually, you won't need to override this method unless you wish to release specific resources that the applet created.

18.7 Passing Strings to Applets

In §9.5, "Command-Line Arguments," you learned how to pass strings to Java applications from a command line. Strings are passed to the `main` method as an array of strings. When the

application starts, the `main` method can use these strings. There is no `main` method in an applet, however, and applets are not run from the command line by the Java interpreter.

How, then, can applets accept arguments? In this section, you will learn how to pass strings to Java applets. To be passed to an applet, a parameter must be declared in the HTML file and must be read by the applet when it is initialized. Parameters are declared using the `<param>` tag. The `<param>` tag must be embedded in the `<applet>` tag and has no end tag. Its syntax is given below:

```
<param name = parametername value = stringvalue />
```

This tag specifies a parameter and its corresponding string value.



Note

No comma separates the parameter name from the parameter value in the HTML code. The HTML parameter names are not case sensitive.

Suppose you want to write an applet to display a message. The message is passed as a parameter. In addition, you want the message to be displayed at a specific location with `x`-coordinate and `y`-coordinate, which are passed as two parameters. The parameters and their values are listed in Table 18.1.

TABLE 18.1 Parameter Names and Values for the `DisplayMessage` Applet

Parameter Name	*Parameter Value*
`MESSAGE`	`"Welcome to Java"`
`X`	`20`
`Y`	`30`

The HTML source file is given in Listing 18.4.

LISTING 18.4 DisplayMessage.html

```
<html>
  <head>
    <title>Passing Strings to Java Applets</title>
  </head>
  <body>
    <p>This applet gets a message from the HTML
       page and displays it.</p>
    <applet
      code = "DisplayMessage.class"
      width = 200
      height = 50
      alt = "You must have a Java 2-enabled browser to view the applet"
    >
      <param name = MESSAGE value = "Welcome to Java" />
      <param name = X value = 20 />
      <param name = Y value = 30 />
    </applet>
  </body>
</html>
```

To read the parameter from the applet, use the following method defined in the `Applet` class:

```
public String getParameter(String parametername);
```

This returns the value of the specified parameter.

The applet is given in Listing 18.5. A sample run of the applet is shown in Figure 18.7.



FIGURE 18.7 The applet displays the message `Welcome to Java` passed from the HTML page.

LISTING 18.5 DisplayMessage.java

```
1  import javax.swing.*;
2
3  public class DisplayMessage extends JApplet {
4    /** Initialize the applet */
5    public void init() {
6      // Get parameter values from the HTML file
7      String message = getParameter("MESSAGE");
8      int x = Integer.parseInt(getParameter("X"));
9      int y = Integer.parseInt(getParameter("Y"));
10
11     // Create a message panel
12     MessagePanel messagePanel = new MessagePanel(message);
13     messagePanel.setXCoordinate(x);
14     messagePanel.setYCoordinate(y);
15
16     // Add the message panel to the applet
17     add(messagePanel);
18   }
19 }
```

getParameter

add to applet

The program gets the parameter values from the HTML in the `init` method. The values are strings obtained using the `getParameter` method (lines 7–9). Because `x` and `y` are `int`, the program uses `Integer.parseInt(string)` to parse a digital string into an `int` value.

If you change `Welcome to Java` in the HTML file to `Welcome to HTML`, and reload the HTML file in the Web browser, you should see `Welcome to HTML` displayed. Similarly, the `x` and `y` values can be changed to display the message in a desired location.



Caution

The `Applet`'s `getParameter` method can be invoked only after an instance of the applet is created. Therefore, this method cannot be invoked in the constructor of the applet class. You should invoke it from the `init` method.

You can add a main method to enable this applet to run standalone. The applet takes the parameters from the HTML file when it runs as an applet and takes the parameters from the command line when it runs standalone. The program, as shown in Listing 18.6, is identical to `DisplayMessage` except for the addition of a new `main` method and of a variable named `isStandalone` to indicate whether it is running as an applet or as an application.

LISTING 18.6 DisplayMessageApp.java

```
1  import javax.swing.*;
2  import java.awt.Font;
3  import java.awt.BorderLayout;
4
5  public class DisplayMessageApp extends JApplet {
6    private String message = "A default message"; // Message to display
7    private int x = 20; // Default x-coordinate
8    private int y = 20; // Default y-coordinate
9
10   /** Determine whether it is an application */
11   private boolean isStandalone = false;                          isStandalone
12
13   /** Initialize the applet */
14   public void init() {
15     if (!isStandalone) {
16       // Get parameter values from the HTML file
17       message = getParameter("MESSAGE");                         applet params
18       x = Integer.parseInt(getParameter("X"));
19       y = Integer.parseInt(getParameter("Y"));
20     }
21
22     // Create a message panel
23     MessagePanel messagePanel = new MessagePanel(message);
24     messagePanel.setFont(new Font("SansSerif", Font.BOLD, 20));
25     messagePanel.setXCoordinate(x);
26     messagePanel.setYCoordinate(y);
27
28     // Add the message panel to the applet
29     add(messagePanel);
30   }
31
32   /** Main method to display a message
33      @param args[0] x-coordinate
34      @param args[1] y-coordinate
35      @param args[2] message
36     */
37   public static void main(String[] args) {
38     // Create a frame
39     JFrame frame = new JFrame("DisplayMessageApp");
40
41     // Create an instance of the applet
42     DisplayMessageApp applet = new DisplayMessageApp();
43
44     // It runs as an application
45     applet.isStandalone = true;                                  standalone
46
47     // Get parameters from the command line
48     applet.getCommandLineParameters(args);                       command params
49
50     // Add the applet instance to the frame
51     frame.add(applet, BorderLayout.CENTER);
52
53     // Invoke applet's init method
54     applet.init();
55     applet.start();
56
57     // Display the frame
58     frame.setSize(300, 300);
```

```
59      frame.setLocationRelativeTo(null); // Center the frame
60      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
61      frame.setVisible(true);
62    }
63
64    /** Get command-line parameters */
65    private void getCommandLineParameters(String[] args) {
66      // Check usage and get x, y and message
67      if (args.length != 3) {
68        System.out.println(
69          "Usage: java DisplayMessageApp x y message");
70        System.exit(0);
71      }
72      else {
73        x = Integer.parseInt(args[0]);
74        y = Integer.parseInt(args[1]);
75        message = args[2];
76      }
77    }
78  }
```

When you run the program as an applet, the `main` method is ignored. When you run it as an application, the `main` method is invoked. Sample runs of the program as an application and as an applet are shown in Figure 18.8.



FIGURE 18.8 The `DisplayMessageApp` class can run as an application and as an applet.

The `main` method creates a `JFrame` object `frame` and creates a `JApplet` object `applet`, then places the applet `applet` into the frame `frame` and invokes its `init` method. The application runs just like an applet.

The `main` method sets `isStandalone true` (line 45) so that it does not attempt to retrieve HTML parameters when the `init` method is invoked.

The `setVisible(true)` method (line 61) is invoked *after* the components are added to the applet, and the applet is added to the frame to ensure that the components will be visible. Otherwise, the components are not shown when the frame starts.

omitting **main** method



Important Pedagogical Note

From now on, all the GUI examples will be created as applets with a `main` method. Thus you will be able to run the program either as an applet or as an application. For brevity, the `main` method is not listed in the text.

18.8 Case Study: Bouncing Ball

This section presents an applet that displays a ball bouncing in a panel. Use two buttons to suspend and resume the movement, and use a scroll bar to control the bouncing speed, as shown in Figure 18.9.

Here are the major steps to complete this example:

1. Create a subclass of `JPanel` named `Ball` to display a ball bouncing, as shown in Listing 18.7.



FIGURE 18.9 The ball's movement is controlled by the Suspend and Resume buttons and the scroll bar.

2. Create a subclass of `JPanel` named `BallControl` to contain the ball with a scroll bar and two control buttons *Suspend* and *Resume*, as shown in Listing 18.8.
3. Create an applet named `BounceBallApp` to contain an instance of `BallControl` and enable the applet to run standalone, as shown in Listing 18.9.

The relationship among these classes is shown in Figure 18.10.

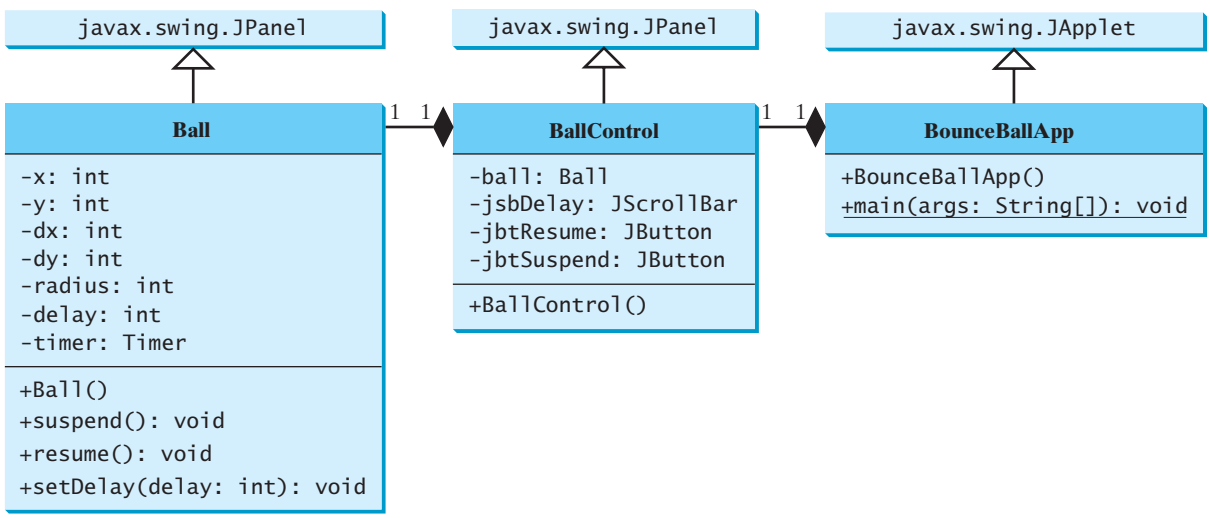


FIGURE 18.10 `BounceBallApp` contains `BallControl`, and `BallControl` contains `Ball`.

LISTING 18.7 Ball.java

```
1  import javax.swing.Timer;
2  import java.awt.*;
3  import javax.swing.*;
4  import java.awt.event.*;
5
6  public class Ball extends JPanel {
7    private int delay = 10;                                          timer delay
8
9    // Create a timer with delay 1000 ms
10   private Timer timer = new Timer(delay, new TimerListener());     create timer
11
12   private int x = 0; private int y = 0; // Current ball position
13   private int radius = 5; // Ball radius
14   private int dx = 2; // Increment on ball's x-coordinate
15   private int dy = 2; // Increment on ball's y-coordinate
16
```

start timer

timer listener

repaint ball

paint ball

```
17   public Ball() {
18     timer.start();
19   }
20
21   private class TimerListener implements ActionListener {
22     /** Handle the action event */
23     public void actionPerformed(ActionEvent e) {
24       repaint();
25     }
26   }
27
28   protected void paintComponent(Graphics g) {
29     super.paintComponent(g);
30
31     g.setColor(Color.red);
32
33     // Check boundaries
34     if (x < radius) dx = Math.abs(dx);
35     if (x > getWidth() - radius) dx = -Math.abs(dx);
36     if (y < radius) dy = Math.abs(dy);
37     if (y > getHeight() - radius) dy = -Math.abs(dy);
38
39     // Adjust ball position
40     x += dx;
41     y += dy;
42     g.fillOval(x - radius, y - radius, radius * 2, radius * 2);
43   }
44
45   public void suspend() {
46     timer.stop(); // Suspend timer
47   }
48
49   public void resume() {
50     timer.start(); // Resume timer
51   }
52
53   public void setDelay(int delay) {
54     this.delay = delay;
55     timer.setDelay(delay);
56   }
57 }
```

The use of `Timer` to control animation was introduced in §16.12, "Animation Using the `Timer` Class." `Ball` extends `JPanel` to display a moving ball. The timer listener implements `ActionListener` to listen for `ActionEvent` (line 21). Line 10 creates a `Timer` for a `Ball`. The timer is started in line 18 when a `Ball` is constructed. The timer fires an `ActionEvent` at a fixed rate. The listener responds in line 24 to repaint the ball to animate ball movement. The center of the ball is at (`x`, `y`), which changes to (`x + dx`, `y + dy`) on the next display (lines 40–41). The `suspend` and `resume` methods (lines 45–51) can be used to stop and start the timer. The `setDelay(int)` method (lines 53–56) sets a new delay.

LISTING 18.8 BallControl.java

button

```
1 import javax.swing.*;
2 import java.awt.event.*;
3 import java.awt.*;
4
5 public class BallControl extends JPanel {
6   private Ball ball = new Ball();
```

```
 7   private JButton jbtSuspend = new JButton("Suspend");
 8   private JButton jbtResume = new JButton("Resume");                scroll bar
 9   private JScrollBar jsbDelay = new JScrollBar();
10
11   public BallControl() {                                            create UI
12     // Group buttons in a panel
13     JPanel panel = new JPanel();
14     panel.add(jbtSuspend);
15     panel.add(jbtResume);
16
17     // Add ball and buttons to the panel
18     ball.setBorder(new javax.swing.border.LineBorder(Color.red));
19     jsbDelay.setOrientation(JScrollBar.HORIZONTAL);
20     ball.setDelay(jsbDelay.getMaximum());
21     setLayout(new BorderLayout());
22     add(jsbDelay, BorderLayout.NORTH);
23     add(ball, BorderLayout.CENTER);
24     add(panel, BorderLayout.SOUTH);
25
26     // Register listeners
27     jbtSuspend.addActionListener(new ActionListener() {             register listener
28       public void actionPerformed(ActionEvent e) {
29         ball.suspend();                                             suspend
30       }
31     });
32     jbtResume.addActionListener(new ActionListener() {              register listener
33       public void actionPerformed(ActionEvent e) {
34         ball.resume();                                              resume
35       }
36     });
37     jsbDelay.addAdjustmentListener(new AdjustmentListener() {       register listener
38       public void adjustmentValueChanged(AdjustmentEvent e) {
39         ball.setDelay(jsbDelay.getMaximum() - e.getValue());        new delay
40       }
41     });
42   }
43 }
```

The `BallControl` class extends `JPanel` to display the ball with a scroll bar and two control buttons. When the *Suspend* button is clicked, the ball's `suspend()` method is invoked to suspend the ball movement (line 29). When the *Resume* button is clicked, the ball's `resume()` method is invoked to resume the ball movement (line 34). The bouncing speed can be changed using the scroll bar.

LISTING 18.9 BounceBallApp.java

```
1 import java.awt.*;
2 import javax.swing.*;
3
4 public class BounceBallApp extends JApplet {
5   public BounceBallApp() {
6     add(new BallControl());                                           add BallControl
7   }
8 }                                                                     main method omitted
```

The `BounceBallApp` class simply places an instance of `BallControl` in the applet. The `main` method is provided in the applet (not displayed in the listing for brevity) so that you can also run it standalone.

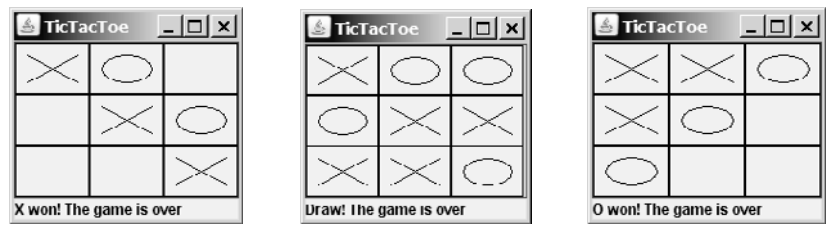
18.9 Case Study: TicTacToe



Video Note
TicTacToe

From the many examples in this and earlier chapters you have learned about objects, classes, arrays, class inheritance, GUI, event-driven programming, and applets. Now it is time to put what you have learned to work in developing comprehensive projects. In this section, you will develop a Java applet with which to play the popular game of TicTacToe.

Two players take turns marking an available cell in a 3 × 3 grid with their respective tokens (either X or O). When one player has placed three tokens in a horizontal, vertical, or diagonal row on the grid, the game is over and that player has won. A draw (no winner) occurs when all the cells on the grid have been filled with tokens and neither player has achieved a win. Figure 18.11 shows the representative sample runs of the example.



(a) The X player won the game (b) Draw-no winners (c) The O player won the game

FIGURE 18.11 Two players play a TicTacToe game.

All the examples you have seen so far show simple behaviors that are easy to model with classes. The behavior of the TicTacToe game is somewhat more complex. To create classes that model the behavior, you need to study and understand the game.

Assume that all the cells are initially empty, and that the first player takes the X token, the second player the O token. To mark a cell, the player points the mouse to the cell and clicks it. If the cell is empty, the token (X or O) is displayed. If the cell is already filled, the player's action is ignored.

From the preceding description, it is obvious that a cell is a GUI object that handles the mouse-click event and displays tokens. Such an object could be either a button or a panel. Drawing on panels is more flexible than drawing on buttons, because on a panel the token (X or O) can be drawn in any size, but on a button it can be displayed only as a text label. Therefore, a panel should be used to model a cell. How do you know the state of the cell (empty, X, or O)? You use a property named `token` of `char` type in the `Cell` class. The `Cell` class is responsible for drawing the token when an empty cell is clicked. So you need to write the code for listening to the `MouseEvent` and for painting the shapes for tokens X and O. The `Cell` class can be defined as shown in Figure 18.12.

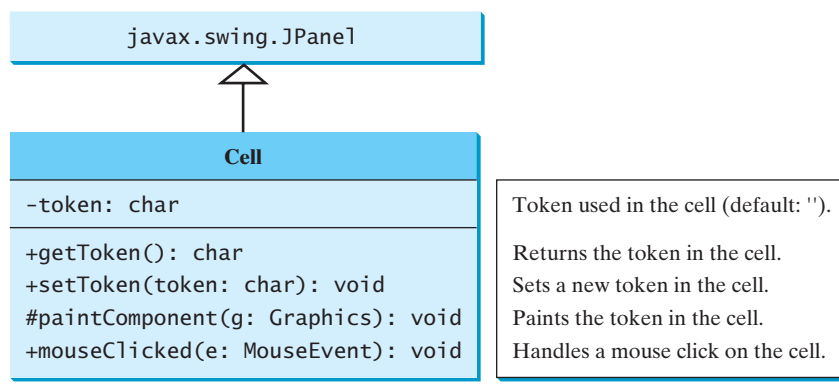

javax.swing.JPanel
Cell
-token: char
+getToken(): char
+setToken(token: char): void
#paintComponent(g: Graphics): void
+mouseClicked(e: MouseEvent): void
Token used in the cell (default: '').
Returns the token in the cell.
Sets a new token in the cell.
Paints the token in the cell.
Handles a mouse click on the cell.

FIGURE 18.12 The `Cell` class paints the token on a cell.

The TicTacToe board consists of nine cells, created using `new Cell[3][3]`. To determine which player's turn it is, you can introduce a variable named `whoseTurn` of `char` type. `whoseTurn` is initially X, then changes to O, and subsequently changes between X and O whenever a new cell is occupied. When the game is over, set `whoseTurn` to `' '`.

How do you know whether the game is over, whether there is a winner, and who the winner, if any, is? You can create a method named `isWon(char token)` to check whether a specified token has won and a method named `isFull()` to check whether all the cells are occupied.

Clearly, two classes emerge from the foregoing analysis. One is the `Cell` class, which handles operations for a single cell; the other is the `TicTacToe` class, which plays the whole game and deals with all the cells. The relationship between these two classes is shown in Figure 18.13.

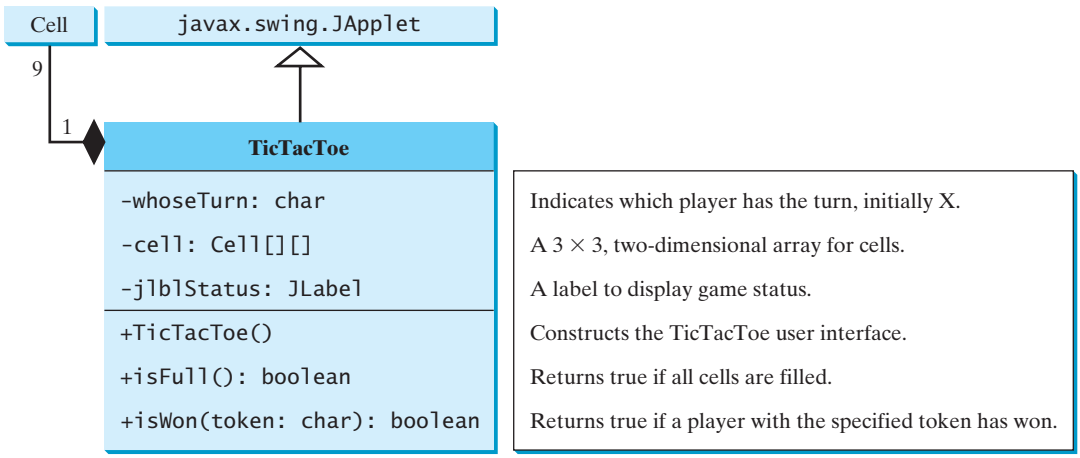


FIGURE 18.13 The `TicTacToe` class contains nine cells.

Since the `Cell` class is only to support the `TicTacToe` class, it can be defined as an inner class in `TicTacToe`. The complete program is given in Listing 18.10:

LISTING 18.10 TicTacToe.java

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  import javax.swing.border.LineBorder;
5
6  public class TicTacToe extends JApplet {                    main class TicTacToe
7    // Indicate which player has a turn; initially it is the X player
8    private char whoseTurn = 'X';
9
10   // Create and initialize cells
11   private Cell[][] cells =  new Cell[3][3];
12
13   // Create and initialize a status label
14   private JLabel jlblStatus = new JLabel("X's turn to play");
15
16   /** Initialize UI */
17   public TicTacToe() {
18     // Panel p to hold cells
19     JPanel p = new JPanel(new GridLayout(3, 3, 0, 0));
20     for (int i = 0; i < 3; i++)
21       for (int j = 0; j < 3; j++)
22         p.add(cells[i][j] = new Cell());
```

```
23
24      // Set line borders on the cells panel and the status label
25      p.setBorder(new LineBorder(Color.red, 1));
26      jlblStatus.setBorder(new LineBorder(Color.yellow, 1));
27
28      // Place the panel and the label to the applet
29      add(p, BorderLayout.CENTER);
30      add(jlblStatus, BorderLayout.SOUTH);
31    }
32
33    /** Determine whether the cells are all occupied */
check isFull
34    public boolean isFull() {
35      for (int i = 0; i < 3; i++)
36        for (int j = 0; j < 3; j++)
37          if (cells[i][j].getToken() == ' ')
38            return false;
39
40      return true;
41    }
42
43    /** Determine whether the player with the specified token wins */
44    public boolean isWon(char token) {
check rows
45      for (int i = 0; i < 3; i++)
46        if ((cells[i][0].getToken() == token)
47            && (cells[i][1].getToken() == token)
48            && (cells[i][2].getToken() == token)) {
49          return true;
50        }
51
check columns
52      for (int j = 0; j < 3; j++)
53        if ((cells[0][j].getToken() ==  token)
54            && (cells[1][j].getToken() == token)
55            && (cells[2][j].getToken() == token)) {
56          return true;
57        }
58
check major diagonal
59      if ((cells[0][0].getToken() == token)
60          && (cells[1][1].getToken() == token)
61          && (cells[2][2].getToken() == token)) {
62        return true;
63      }
64
check subdiagonal
65      if ((cells[0][2].getToken() == token)
66          && (cells[1][1].getToken() == token)
67          && (cells[2][0].getToken() == token)) {
68        return true;
69      }
70
71      return false;
72    }
73
74    // An inner class for a cell
inner class Cell
75    public class Cell extends JPanel {
76      // Token used for this cell
77      private char token = ' ';
78
79      public Cell() {
80        setBorder(new LineBorder(Color.black, 1)); // Set cell's border
register listener
81        addMouseListener(new MyMouseListener());  // Register listener
82      }
```

```
83
84      /** Return token */
85      public char getToken() {
86        return token;
87      }
88
89      /** Set a new token */
90      public void setToken(char c) {
91        token = c;
92        repaint();
93      }
94
95      /** Paint the cell */
96      protected void paintComponent(Graphics g) {                    paint cell
97        super.paintComponent(g);
98
99        if (token == 'X') {
100         g.drawLine(10, 10, getWidth() - 10, getHeight() - 10);
101         g.drawLine(getWidth() - 10, 10, 10, getHeight() - 10);
102       }
103       else if (token == 'O') {
104         g.drawOval(10, 10, getWidth() - 20, getHeight() - 20);
105       }
106     }
107
108     private class MyMouseListener extends MouseAdapter {           listener class
109       /** Handle mouse click on a cell */
110       public void mouseClicked(MouseEvent e) {
111         // If cell is empty and game is not over
112         if (token == ' ' && whoseTurn != ' ') {
113           setToken(whoseTurn); // Set token in the cell
114
115           // Check game status
116           if (isWon(whoseTurn)) {
117             jlblStatus.setText(whoseTurn + " won! The game is over");
118             whoseTurn = ' '; // Game is over
119           }
120           else if (isFull()) {
121             jlblStatus.setText("Draw! The game is over");
122             whoseTurn = ' '; // Game is over
123           }
124           else {
125             // Change the turn
126             whoseTurn = (whoseTurn == 'X') ? 'O': 'X';
127             // Display whose turn
128             jlblStatus.setText(whoseTurn + "'s turn");
129           }
130         }
131       }
132     }
133   }
134 }                                                                  main method omitted
```

The `TicTacToe` class initializes the user interface with nine cells placed in a panel of `GridLayout` (lines 19–22). A label named `jlblStatus` is used to show the status of the game (line 14). The variable `whoseTurn` (line 8) is used to track the next type of token to be placed in a cell. The methods `isFull` (lines 34-41) and `isWon` (lines 44–72) are for checking the status of the game.

Since `Cell` is an inner class in `TicTacToe`, the variable (`whoseTurn`) and methods (`isFull` and `isWon`) defined in `TicTacToe` can be referenced from the `Cell` class. The inner class makes

programs simple and concise. If `Cell` were not declared as an inner class of `TicTacToe`, you would have to pass an object of `TicTacToe` to `Cell` in order for the variables and methods in `TicTacToe` to be used in `Cell`. You will rewrite the program without using an inner class in Exercise 18.6.

The listener for `MouseEvent` is registered for the cell (line 81). If an empty cell is clicked and the game is not over, a token is set in the cell (line 113). If the game is over, `whoseTurn` is set to `' '` (lines 118, 122). Otherwise, `whoseTurn` is alternated to a new turn (line 126).



Tip

incremental development and testing

Use an incremental approach in developing and testing a Java project of this kind. The foregoing program can be divided into five steps:

1. Lay out the user interface and display a fixed token X on a cell.
2. Enable the cell to display a fixed token X upon a mouse click.
3. Coordinate between the two players so as to display tokens X and O alternately.
4. Check whether a player wins, or whether all the cells are occupied without a winner.
5. Implement displaying a message on the label upon each move by a player.

18.10 Locating Resources Using the URL Class

You have used the `ImageIcon` class to create an icon from an image file and used the `setIcon` method or the constructor to place the icon in a GUI component, such as a button or a label. For example, the following statements create an `ImageIcon` and set it on a `JLabel` object `jlbl`:

```
ImageIcon imageIcon = new ImageIcon("c:\\book\\image\\us.gif");
jlbl.setIcon(imageIcon);
```

This approach presents a problem. The file location is fixed, because it uses the absolute file path on the Windows platform. As a result, the program cannot run on other platforms and cannot run as an applet. Assume that `image/us.gif` is under the class directory. You can circumvent this problem by using a relative path as follows:

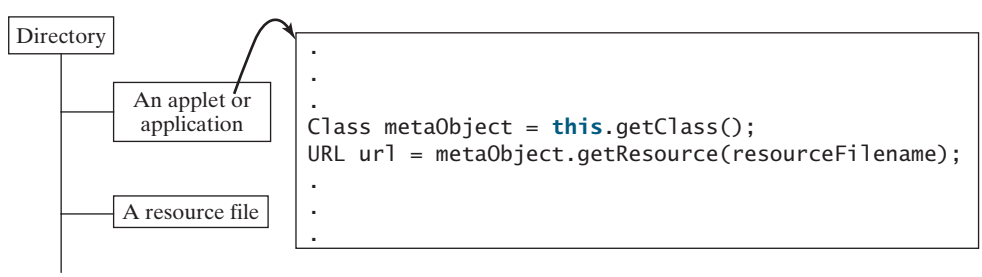
```
ImageIcon imageIcon = new ImageIcon("image/us.gif");
```

This works fine with Java applications on all platforms but not with Java applets, because applets cannot load local files. To enable it to work with both applications and applets, you need to locate the file's URL.

why URL class?

The `java.net.URL` class can be used to identify files (image, audio, text, and so on) on the Internet. In general, a URL (Uniform Resource Locator) is a pointer to a "resource" on a local machine or a remote host. A resource can be something as simple as a file or a directory.

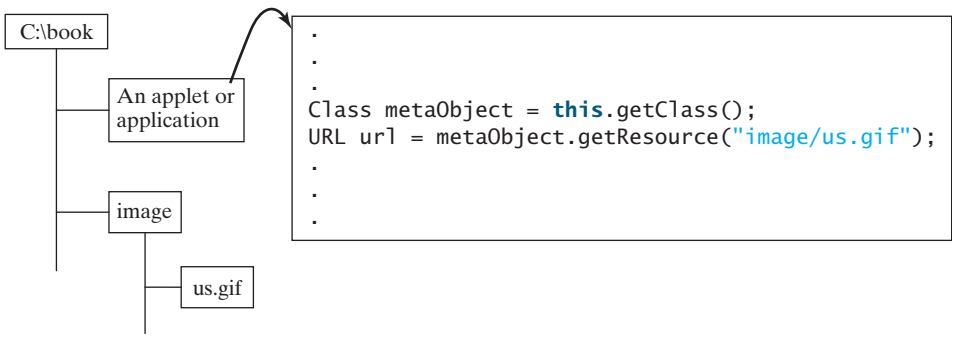
A URL for a file can also be accessed from a class in a way that is independent of the location of the file, as long as the file is located in the class directory. Recall that the class directory is where the class is stored. To obtain the URL of a resource file, use the following statements in an applet or application:



The `getClass()` method returns an instance of the `java.lang.Class` class. This instance is automatically created by the JVM for every class loaded into the memory. This instance, also known as a *meta object*, contains the information about the class file such as class name,

meta object

constructors, and methods. You can obtain the URL of a file in the class path by invoking the `getResource(filename)` method on the meta object. For example, if the class file is in `c:\book`, the following statements obtain a URL for `c:\book\image\us.gif`.



You can now create an `ImageIcon` using

```
ImageIcon imageIcon = new ImageIcon(url);
```

Listing 18.11 gives the code that displays an image from `image/us.gif` in the class directory. The file `image/us.gif` is under the class directory, and its URL is obtained using the `getResource` method (line 5). A label with an image icon is created in line 6. The image icon is obtained from the URL.

LISTING 18.11 DisplayImageWithURL.java

```
1 import javax.swing.*;
2
3 public class DisplayImageWithURL extends JApplet {
4   public DisplayImageWithURL() {
5     java.net.URL url = this.getClass().getResource("image/us.gif");
6     add(new JLabel(new ImageIcon(url)));
7   }
8 }
```

get image URL
create a label

main method omitted

If you replace the code in lines 5–6 with the following code,

```
add(new JLabel(new ImageIcon("image/us.gif")));
```

you can still run the program standalone, but not from a browser.

18.11 Playing Audio in Any Java Program

There are several formats for audio files. Java programs can play audio files in the WAV, AIFF, MIDI, AU, and RMF formats.

To play an audio file in Java (application or applet), first create an *audio clip object* for the file. The audio clip is created once and can be played repeatedly without reloading the file. To create an audio clip, use the static method `newAudioClip()` in the `java.applet.Applet` class:

```
AudioClip audioClip = Applet.newAudioClip(url);
```

Audio originally could be played only from Java applets. For this reason, the `AudioClip` interface is in the `java.applet` package. Since JDK 1.2, audio can be played in any Java program.

The following statements, for example, create an `AudioClip` for the `beep.au` audio file in the class directory:

```
Class metaObject = this.getClass();
URL url = metaObject.getResource("beep.au");
AudioClip audioClip = Applet.newAudioClip(url);
```

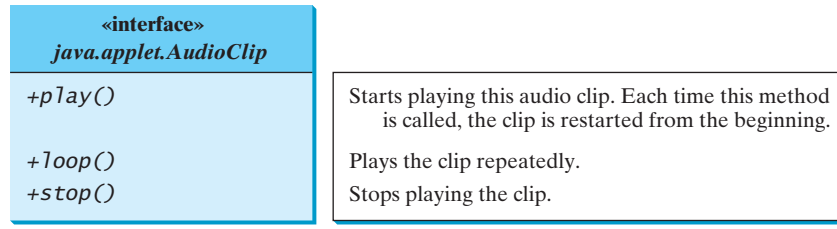


FIGURE 18.14 The `AudioClip` interface provides the methods for playing sound.

To manipulate a sound for an audio clip, use the `play()`, `loop()`, and `stop()` methods in `java.applet.AudioClip`, as shown in Figure 18.14.

Listing 18.12 gives the code that displays the Danish flag and plays the Danish national anthem repeatedly. The image file `image/denmark.gif` and audio file `audio/denmark.mid` are stored under the class directory. Line 12 obtains the audio file URL. Line 13 creates an audio clip for the file. Line 14 repeatedly plays the audio.

LISTING 18.12 DisplayImagePlayAudio.java

```
1  import javax.swing.*;
2  import java.net.URL;
3  import java.applet.*;
4
5  public class DisplayImagePlayAudio extends JApplet {
6    private AudioClip audioClip;
7
8    public DisplayImagePlayAudio() {
9      URL urlForImage = getClass().getResource("image/denmark.gif");
10     add(new JLabel(new ImageIcon(urlForImage)));
11
12     URL urlForAudio = getClass().getResource("audio/denmark.mid");
13     audioClip = Applet.newAudioClip(urlForAudio);
14     audioClip.loop();
15   }
16
17   public void start() {
18     if (audioClip != null) audioClip.loop();
19   }
20
21   public void stop() {
22     if (audioClip != null) audioClip.stop();
23   }
24 }
```

get image URL
create a label

get audio URL
create an audio clip
play audio repeatedly

start audio

stop audio

main method omitted

The `stop` method (lines 21–23) stops the audio when the applet is not displayed, and the `start` method (lines 17–19) restarts the audio when the applet is redisplayed. Try to run this applet from a browser and observe the effect without the `stop` and `start` methods.

Run this program standalone from the main method and from a Web browser to test it. Recall that, for brevity, the `main` method in all applets is not printed in the listing.

18.12 Case Study: Multimedia Animations

This case study presents a multimedia animation with images and audio. The images are for seven national flags, named `flag0.gif`, `flag1.gif`, . . . , `flag6.gif` for Denmark, Germany, China, India, Norway, U.K., and U.S. They are stored under the `image` directory in the class path. The audio consists of national anthems for these seven nations, named

anthem0.mid, **anthem1.mid**, . . . , and **anthem6.mid**. They are stored under the **audio** directory in the class path.

The program presents the nations, starting from the first one. For each nation, it displays its flag and plays its anthem. When the audio for a nation finishes, the next nation is presented, and so on. After the last nation is presented, the program starts to present all the nations again. You may suspend animation by clicking the *Suspend* button and resume it by clicking the *Resume* button, as shown in Figure 18.15. You can also directly select a nation from a combo box.

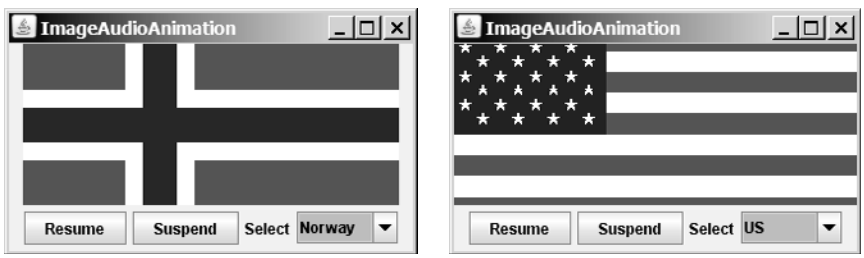


FIGURE 18.15 The applet displays a sequence of images and plays audio.

The program is given in Listing 18.13. A timer is created to control the animation (line 15). The timer delay for each presentation is the play time for the anthem. You can find the play time for an audio file using RealPlayer or Windows Media. The delay times are stored in an array named **delays** (lines 13–14). The delay time for the first audio file (the Danish anthem) is 48 seconds.

LISTING 18.13 ImageAudioAnimation.java



Video Note
Audio and image

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  import java.applet.*;
5
6  public class ImageAudioAnimation extends JApplet {
7    private final static int NUMBER_OF_NATIONS = 7;
8    private int current = 0;
9    private ImageIcon[] icons = new ImageIcon[NUMBER_OF_NATIONS];          image icons
10   private AudioClip[] audioClips = new AudioClip[NUMBER_OF_NATIONS];     audio clips
11   private AudioClip currentAudioClip;                                   current audio clip
12
13   private int[] delays =                                                audio play time
14     {48000, 54000, 59000, 54000, 59000, 31000, 68000};
15   private Timer timer = new Timer(delays[0], new TimerListener());      timer
16
17   private JLabel jlblImageLabel = new JLabel();                         GUI components
18   private JButton jbtResume = new JButton("Resume");
19   private JButton jbtSuspend = new JButton("Suspend");
20   private JComboBox jcboNations = new JComboBox(new Object[]
21     {"Denmark", "Germany", "China", "India", "Norway", "UK", "US"});
22
23   public ImageAudioAnimation() {
24     // Load image icons and audio clips
25     for (int i = 0; i < NUMBER_OF_NATIONS; i++) {
26       icons[i] = new ImageIcon(getClass().getResource(                  create icons
27         "image/flag" + i + ".gif"));
28       audioClips[i] = Applet.newAudioClip(                              create audio clips
```

```
29          getClass().getResource("audio/anthem" + i + ".mid"));
30      }
31
create UI 32      JPanel panel = new JPanel();
33      panel.add(jbtResume);
34      panel.add(jbtSuspend);
35      panel.add(new JLabel("Select"));
36      panel.add(jcboNations);
37      add(jlblImageLabel, BorderLayout.CENTER);
38      add(panel, BorderLayout.SOUTH);
39
register listener 40      jbtResume.addActionListener(new ActionListener() {
41        public void actionPerformed(ActionEvent e) {
start animation 42          start();
43        }
44      });
register listener 45      jbtSuspend.addActionListener(new ActionListener() {
46        public void actionPerformed(ActionEvent e) {
stop animation 47          stop();
48        }
49      });
register listener 50      jcboNations.addActionListener(new ActionListener() {
51        public void actionPerformed(ActionEvent e) {
52          stop();
select a nation 53          current = jcboNations.getSelectedIndex();
present a nation 54          presentNation(current);
55          timer.start();
56        }
57      });
58
59      timer.start();
60      jlblImageLabel.setIcon(icons[0]);
61      jlblImageLabel.setHorizontalAlignment(JLabel.CENTER);
62      currentAudioClip = audioClips[0];
63      currentAudioClip.play();
64    }
65
66    private class TimerListener implements ActionListener {
67      public void actionPerformed(ActionEvent e) {
68        current = (current + 1) % NUMBER_OF_NATIONS;
69        presentNation(current);
70      }
71    }
72
73    private void presentNation(int index) {
74      jlblImageLabel.setIcon(icons[index]);
75      jcboNations.setSelectedIndex(index);
76      currentAudioClip = audioClips[index];
77      currentAudioClip.play();
set a new delay 78      timer.setDelay(delays[index]);
79    }
80
81    public void start() {
82      timer.start();
83      currentAudioClip.play();
84    }
```

```
85
86   public void stop() {
87     timer.stop();
88     currentAudioClip.stop();
89   }
90 }
```

stop audio clip

main method omitted

A label is created in line 17 to display a flag image. An array of flag images for seven nations is created in lines 26–27. An array of audio clips is created in lines 28–29. Each audio clip is created for an audio file through the URL of the current class. The audio files are stored in the same directory with the applet class file.

The combo box for country names is created in lines 20–21. When a new country name in the combo box is selected, the current presentation is stopped and a new selected nation is presented (lines 52–55).

The `presentNation(index)` method (lines 73–79) presents a nation with the specified index. It sets a new image in the label (line 74), synchronizes with the combo box by setting the selected index (line 75), plays the new audio, and sets a new delay time (line 78).

The applet's `start` and `stop` methods are overridden to resume and suspend the animation (lines 81–89).

Key Terms

applet 616
applet container 620
archive 616
HTML 616
tag 616
signed applet 618

Chapter Summary

1. `JApplet` is a subclass of `Applet`. It is used for developing Java applets with Swing components.
2. The applet bytecode must be specified, using the `<applet>` tag in an HTML file to tell the Web browser where to find the applet. The applet can accept string parameters from HTML using the `<param>` tag.
3. The applet container controls and executes applets through the `init`, `start`, `stop`, and `destroy` methods in the `Applet` class.
4. When an applet is loaded, the applet container creates an instance of it by invoking its no-arg constructor. The `init` method is invoked after the applet is created. The `start` method is invoked after the `init` method. It is also called whenever the applet becomes active again after the page containing the applet is revisited. The `stop` method is invoked when the applet becomes inactive.
5. The `destroy` method is invoked when the browser exits normally to inform the applet that it is no longer needed and should release any resources it has allocated. The `stop` method is always called before the `destroy` method.

6. The procedures for writing applications and writing applets are very similar. An applet can easily be converted into an application, and vice versa. Moreover, an applet can be written with a main method to run standalone.

7. You can pass arguments to an applet using the `param` attribute in the applet's tag in HTML. To retrieve the value of the parameter, invoke the `getParameter(paramName)` method.

8. The `Applet`'s `getParameter` method can be invoked only after an instance of the applet is created. Therefore, this method cannot be invoked in the constructor of the applet class. You should invoke this method from the `init` method.

9. You learned how to incorporate images and audio in Java applications and applets. To load audio and images for Java applications and applets, you have to create a URL for the audio and image. You can create a `URL` from a file under the class directory.

10. To play an audio, create an audio clip from the `URL` for the audio source. You can use the `AudioClip`'s `play()` method to play it once, the `loop()` method to play it repeatedly, and the `stop()` method to stop it.

Review Questions

Sections 18.2–18.6

18.1 Is every applet an instance of `java.applet.Applet`? Is every applet an instance of `javax.swing.JApplet`?

18.2 Describe the `init()`, `start()`, `stop()`, and `destroy()` methods in the `Applet` class.

18.3 How do you add components to a `JApplet`? What is the default layout manager of the content pane of `JApplet`?

18.4 Why does the applet in (a) below display nothing? Why does the applet in (b) have a runtime `NullPointerException` on the highlighted line?

```
import javax.swing.*;

public class WelcomeApplet extends JApplet {
  public void WelcomeApplet() {
    JLabel jlblMessage =
      new JLabel("It is Java");
  }
}
```

(a)

```
import javax.swing.*;

public class WelcomeApplet extends JApplet {
  private JLabel jlblMessage;

  public WelcomeApplet() {
    JLabel jlblMessage =
      new JLabel("It is Java");
  }

  public void init() {
    add(jlblMessage);
  }
}
```

(b)

18.5 Describe the `<applet>` HTML tag. How do you pass parameters to an applet?

18.6 Where is the `getParameter` method defined?

18.7 What is wrong if the `DisplayMessage` applet is revised as follows?

```
public class DisplayMessage extends JApplet {
  /** Initialize the applet */
  public DisplayMessage() {
    // Get parameter values from the HTML file
    String message = getParameter("MESSAGE");
    int x =
      Integer.parseInt(getParameter("X"));
    int y =
      Integer.parseInt(getParameter("Y"));

    // Create a message panel
    MessagePanel messagePanel =
      new MessagePanel(message);
    messagePanel.setXCoordinate(x);
    messagePanel.setYCoordinate(y);

    // Add the message panel to the applet
    add(messagePanel);
  }
}
```

(a) Revision 1

```
public class DisplayMessage extends JApplet {
  private String message;
  private int x;
  private int y;

  /** Initialize the applet */
  public void init() {
    // Get parameter values from the HTML file
    message = getParameter("MESSAGE");
    x = Integer.parseInt(getParameter("X"));
    y = Integer.parseInt(getParameter("Y"));
  }

  public DisplayMessage() {
    // Create a message panel
    MessagePanel messagePanel =
      new  MessagePanel(message);
    messagePanel.setXCoordinate(x);
    messagePanel.setYCoordinate(y);

    // Add the message panel to the applet
    add(messagePanel);
  }
}
```

(b) Revision 2

18.8 What are the differences between applications and applets? How do you run an application, and how do you run an applet? Is the compilation process different for applications and applets? List some security restrictions on applets.

18.9 Can you place a frame in an applet?

18.10 Can you place an applet in a frame?

18.11 Delete `super.paintComponent(g)` on line 97 in TicTacToe.java in Listing 18.5 and run the program to see what happens.

Sections 18.9–18.10

18.12 How do you create a `URL` object for the file image/us.gif in the class directory?

18.13 How do you create an `ImageIcon` from the file image/us.gif in the class directory?

Section 18.11

18.14 What types of audio files are used in Java?

18.15 How do you create an audio clip from a file anthem/us.mid in the class directory?

18.16 How do you play, repeatedly play, and stop an audio clip?

Programming Exercises



Pedagogical Note

For every applet in the exercise, add a main method to enable it to run standalone.

run standalone

Sections 18.2–18.6

18.1 (*Converting applications to applets*) Convert Listing 17.2, ButtonDemo.java, into an applet.

18.2* (*Passing strings to applets*) Rewrite Listing 18.5, DisplayMessage.java to display a message with a standard color, font, and size. The `message`, `x`, `y`, `color`, `fontname`, and `fontsize` are parameters in the `<applet>` tag, as shown below:

```
<applet
  code = "Exercise18_2.class"
  width = 200
  height = 50
  alt = "You must have a Java-enabled browser to view the applet"
>
  <param name = MESSAGE value = "Welcome to Java" />
  <param name = X value = 40 />
  <param name = Y value = 50 />
  <param name = COLOR value = "red" />
  <param name = FONTNAME value = "Monospaced" />
  <param name = FONTSIZE value = 20 />
</applet>
```

18.3 (*Loan calculator*) Write an applet for calculating loan payment, as shown in Figure 18.16. The user can enter the interest rate, the number of years, and the loan amount and click the *Compute Payment* button to display the monthly payment and total payment.

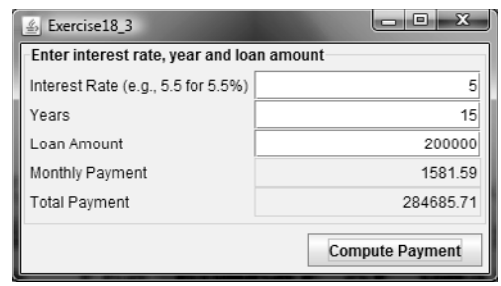
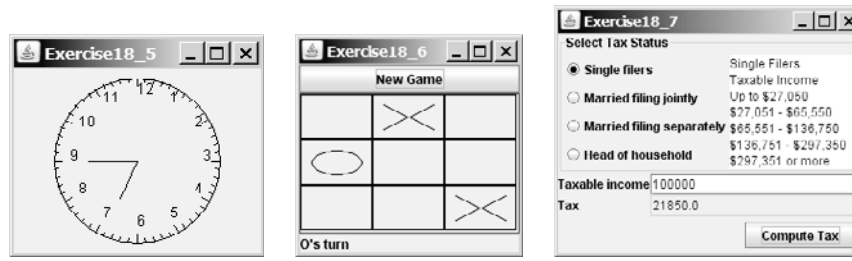


FIGURE 18.16 The applet computes the loan payment.

18.4* (*Converting applications to applets*) Rewrite ClockAnimation in Listing 16.12 as an applet and enable it to run standalone.

18.5** (*Game: a clock learning tool*) Develop a clock applet to show a first-grade student how to read a clock. Modify Exercise 15.19 to display a detailed clock with an hour hand and a minute hand in an applet, as shown in Figure 18.17(a). The hour and minute values are randomly generated. The hour is between `0` and `11`, and the minute is `0`, `15`, `30`, or `45`. Upon a mouse click, a new random time is displayed on the clock.



(a) (b) (c)

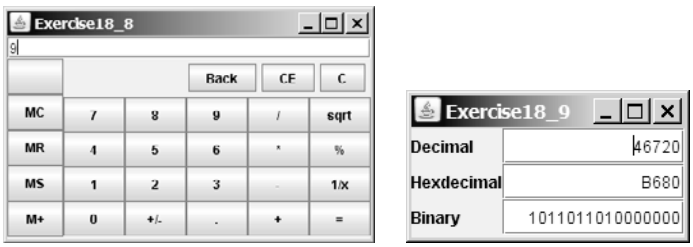
FIGURE 18.17 (a) Upon a mouse click on the clock, the clock time is randomly displayed. (b) The *New Game* button starts a new game. (c) The tax calculator computes the tax for the specified taxable income and tax status.

18.6** (*Game: TicTacToe*) Rewrite the program in §18.9, "Case Study: TicTacToe," with the following modifications:

- Declare `Cell` as a separate class rather than an inner class.
- Add a button named *New Game*, as shown in Figure 18.17(b). The *New Game* button starts a new game.

18.7** (*Financial application: tax calculator*) Create an applet to compute tax, as shown in Figure 18.17(c). The applet lets the user select the tax status and enter the taxable income to compute the tax based on the 2001 federal tax rates, as shown in Exercise 10.8.

18.8*** (*Creating a calculator*) Use various panels of `FlowLayout`, `GridLayout`, and `BorderLayout` to lay out the following calculator and to implement addition (`+`), subtraction (`-`), division (`/`), square root (`sqrt`), and modulus (`%`) functions (see Figure 18.18(a)).



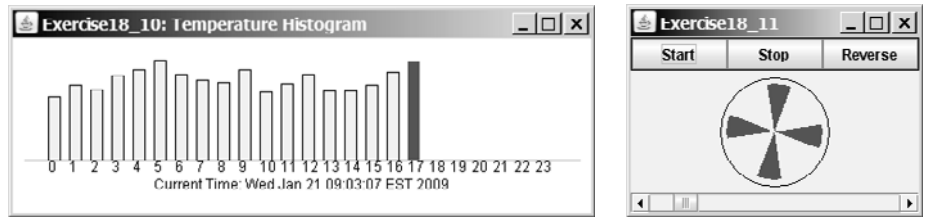
(a) (b)

FIGURE 18.18 (a) Exercise 18.8 is a Java implementation of a popular calculator. (b) Exercise 18.9 converts between decimal, hex, and binary numbers.

18.9* (*Converting numbers*) Write an applet that converts between decimal, hex, and binary numbers, as shown in Figure 18.18(b). When you enter a decimal value on the decimal-value text field and press the *Enter* key, its corresponding hex and binary numbers are displayed in the other two text fields. Likewise, you can enter values in the other fields and convert them accordingly.

18.10** (*Repainting a partial area*) When you repaint the entire viewing area of a panel, sometimes only a tiny portion of the viewing area is changed. You can improve the performance by repainting only the affected area, but do not invoke `super.paintComponent(g)` when repainting the panel, because this will cause the entire viewing area to be cleared. Use this approach to write an applet to display the temperatures of each hour during the last 24 hours in a histogram. Suppose that temperatures between 50 and 90 degrees Fahrenheit are obtained randomly and are updated every hour. The temperature of the current hour needs to be redisplayed, while the others remain unchanged. Use a unique color to highlight the temperature for the current hour (see Figure 18.19(a)).

18.11** (*Simulation: a running fan*) Write a Java applet that simulates a running fan, as shown in Figure 18.19(b). The buttons *Start*, *Stop*, and *Reverse* control the fan. The scrollbar controls the fan's speed. Create a class named `Fan`, a subclass of `JPanel`, to display the fan. This class also contains the methods to suspend and resume the fan, set its speed, and reverse its direction. Create a class named `FanControl` that contains a fan, and three buttons and a scroll bar to control the fan. Create a Java applet that contains an instance of `FanControl`.


Exercise18_10: Temperature Histogram
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
Current Time: Wed Jan 21 09:03:07 EST 2009
Exercise18_11
Start
Stop
Reverse

(a) (b)

FIGURE 18.19 (a) The histogram displays the average temperature of every hour in the last 24 hours. (b) The program simulates a running fan.

18.12** (*Controlling a group of fans*) Write a Java applet that displays three fans in a group, with control buttons to start and stop all of them, as shown in Figure 18.20.



FIGURE 18.20 The program runs and controls a group of fans.

18.13** (*Creating an elevator simulator*) Write an applet that simulates an elevator going up and down (see Figure 18.21). The buttons on the left indicate the floor where the passenger is now located. The passenger must click a button on the left to request that the elevator come to his or her floor. On entering the elevator, the passenger clicks a button on the right to request that it go to the specified floor.



FIGURE 18.21 The program simulates elevator operations.

18.14* (*Controlling a group of clocks*) Write a Java applet that displays three clocks in a group, with control buttons to start and stop all of them, as shown in Figure 18.22.



Video Note
Control a group of clocks

Sections 18.10–18.12

18.15* (*Enlarging and shrinking an image*) Write an applet that will display a sequence of images from a single image file in different sizes. Initially, the viewing area for this image has a width of 300 and a height of 300. Your program should continuously shrink the viewing area by 1 in width and 1 in height until it reaches a width of 50 and a height of 50. At that point, the viewing area



FIGURE 18.22 Three clocks run independently with individual control and group control.

should continuously enlarge by 1 in width and 1 in height until it reaches a width of 300 and a height of 300. The viewing area should shrink and enlarge (alternately) to create animation for the single image.

18.16* (*Simulating a stock ticker*) Write a Java applet that displays a stock-index ticker (see Figure 18.23). The stock-index information is passed from the `<param>` tag in the HTML file. Each index has four parameters: Index Name (e.g., S&P 500), Current Time (e.g., 15:54), the index from the previous day (e.g., 919.01), and Change (e.g., 4.54). Use at least five indexes, such as Dow Jones, S&P 500, NASDAQ, NIKKEI, and Gold & Silver Index. Display positive changes in green and negative changes in red. The indexes move from right to left in the applet's viewing area. The applet freezes the ticker when the mouse button is pressed; it moves again when the mouse button is released.

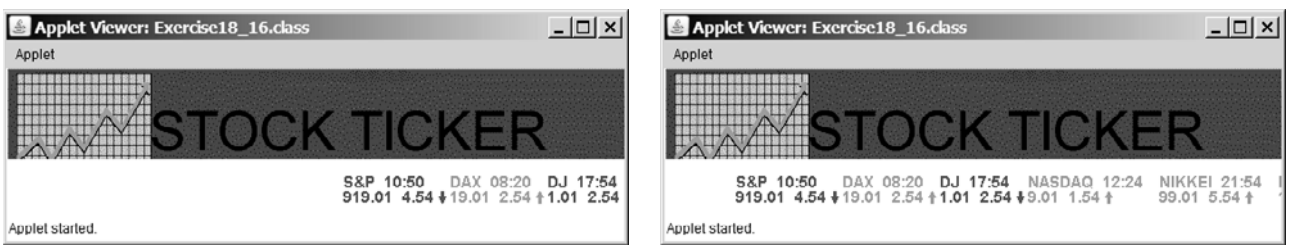
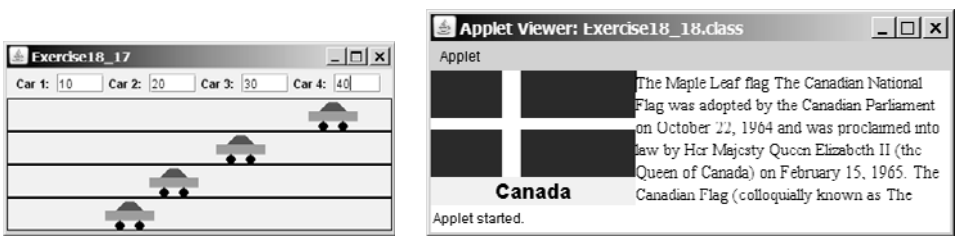

Applet Viewer: Exercise18_16.class
Applet
STOCK TICKER
S&P 10:50 DAX 08:20 DJ 17:54
919.01 4.54 19.01 2.54 1.01 2.54
Applet started.
Applet Viewer: Exercise18_16.class
Applet
STOCK TICKER
S&P 10:50 DAX 08:20 DJ 17:54 NASDAQ 12:24 NIKKEI 21:54
919.01 4.54 19.01 2.54 1.01 2.54 9.01 1.54 99.01 5.54
Applet started.

FIGURE 18.23 The program displays a stock-index ticker.

18.17** (*Racing cars*) Write an applet that simulates four cars racing, as shown in Figure 18.24(a). You can set the speed for each car, with 1 being the highest.


Exercise18_17
Car 1: 10
Car 2: 20
Car 3: 30
Car 4: 40
Applet Viewer: Exercise18_18.class
Applet
Canada
The Maple Leaf flag The Canadian National Flag was adopted by the Canadian Parliament on October 22, 1964 and was proclaimed into law by Her Majesty Queen Elizabeth II (the Queen of Canada) on February 15, 1965. The Canadian Flag (colloquially known as The
Applet started.

(a) (b)

FIGURE 18.24 (a) You can set the speed for each car. (b) This applet shows each country's flag, name, and description, one after another, and reads the description that is currently shown.

18.18** (*Showing national flags*) Write an applet that introduces national flags, one after the other, by presenting each one's photo, name, and description (see Figure 18.24(b)) along with audio that reads the description.

Suppose your applet displays the flags of eight countries. Assume that the photo image files, named **flag0.gif**, **flag1.gif**, and so on, up to **flag7.gif**, are stored in a subdirectory named **image** in the applet's directory. The length of each audio is less than 10 seconds. Assume that the name and description of each country's flag are passed from the HTML using the parameters `name0`, `name1`, ..., `name7`, and `description0`, `description1`, ..., and `description7`. Pass the number of countries as an HTML parameter using `numberOfCountries`. Here is an example:

```
<param name = "numberOfCountries" value = 8>
<param name = "name0" value = "Canada">
<param name = "description0" value = "The Maple Leaf flag
The Canadian National Flag was adopted by the Canadian
Parliament on October 22, 1964 and was proclaimed into law
by Her Majesty Queen Elizabeth II (the Queen of Canada) on
February 15, 1965. The Canadian Flag (colloquially known
as The Maple Leaf Flag) is a red flag of the proportions
two by length and one by width, containing in its center a
white square, with a single red stylized eleven-point
maple leaf centered in the white square.">
```

Hint: Use the `DescriptionPanel` class to display the image, name, and the text. The `DescriptionPanel` class was introduced in Listing 17.6.

18.19* (*Bouncing balls*) The example in §18.8 simulates a bouncing ball. Extend the example to allow multiple balls, as shown in Figure 18.25(a). You may use the *+1* or *−1* button to increase or decrease the number of the balls, and use the *Suspend* and *Resume* buttons to freeze the balls or resume bouncing. For each ball, assign a random color.



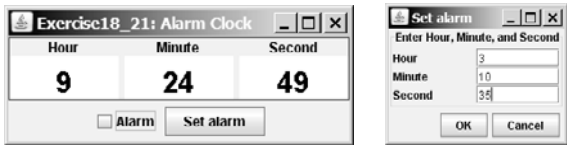
(a) (b)

FIGURE 18.25 (a) The applet allows you to add or remove bouncing balls. (b) Click *Play* to play an audio clip once, click *Loop* to play an audio repeatedly, and click *Stop* to terminate playing.

18.20* (*Playing, looping, and stopping a sound clip*) Write an applet that meets the following requirements:

- Get an audio file. The file is in the class directory.
- Place three buttons labeled *Play*, *Loop*, and *Stop*, as shown in Figure 18.25(b).
- If you click the *Play* button, the audio file is played once. If you click the *Loop* button, the audio file keeps playing repeatedly. If you click the *Stop* button, the playing stops.
- The applet can run as an application.

18.21** (*Creating an alarm clock*) Write an applet that will display a digital clock with a large display panel that shows hour, minute, and second. This clock should allow the user to set an alarm. Figure 18.26(a) shows an example of such a clock. To turn on the alarm, check the *Alarm* check box. To specify the alarm time, click the *Set alarm* button to display a new frame, as shown in Figure 18.26(b). You can set the alarm time in the frame.



(a) (b)

FIGURE 18.26 The program displays current hour, minute, and second and enables you to set an alarm.

18.22** (*Creating an image animator with audio*) Create animation using the applet (see Figure 18.27) to meet the following requirements:

- Allow the user to specify the animation speed. The user can enter the speed in a text field.
- Get the number of frames and the image file-name prefix from the user. For example, if the user enters **n** for the number of frames and **L** for the image prefix, then the files are **L1**, **L2**, and so on, to **L*n***. Assume that the images are stored in the **image** directory, a subdirectory of the applet's directory.
- Allow the user to specify an audio file name. The audio file is stored in the same directory as the applet. The sound is played while the animation runs.

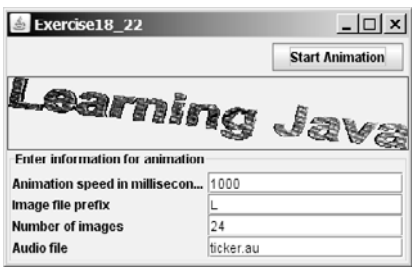


FIGURE 18.27 This applet lets the user select image files, audio file, and animation speed.

18.23** (*Simulation: raising flag and playing anthem*) Create an applet that displays a flag rising up, as shown in Figure 15.1. As the national flag rises, play the national anthem. (You may use a flag image and anthem audio file from Listing 18.13.)

Comprehensive

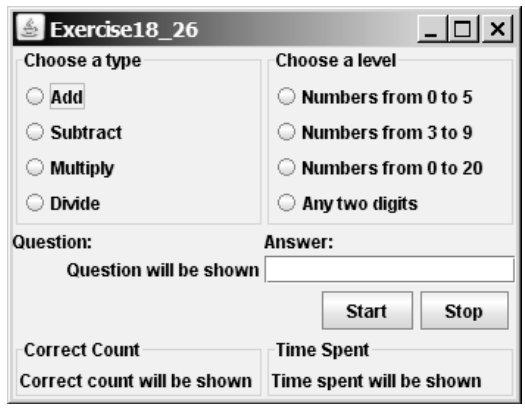
18.24** (*Game: guessing birthdays*) Listing 3.3, GuessBirthday.java, gives a program for guessing a birthday. Create an applet for guessing birthdays as shown in Figure 18.28. The applet prompts the user to check whether the date is in any of the five sets. The date is displayed in the text field upon clicking the *Guess Birthday* button.

18.25*** (*Game: Sudoku*) §7.7 introduced the Sudoku problem. Write a program that lets the user enter the input from the text fields in an applet, as shown in Figure 18.1. Clicking the *Solve* button displays the result.

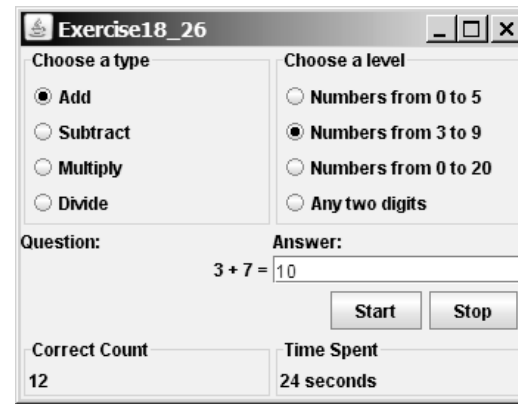


FIGURE 18.28 This applet guesses the birthdays.

18.26* (*Game: math quiz*) Listing 3.1, AdditionQuiz.java, and Listing 3.4, SubtractionQuiz.java, generate and grade Math quizzes. Write an applet that allows the user to select a question type and difficulty level, as shown in Figure 18.29(a). When the user clicks the *Start* button, the program begins to generate a question. After the user enters an answer with the *Enter* key, a new question is displayed. When the user clicks the *Start* button, the elapse time is displayed. The time is updated every second until the *Stop* button is clicked. The correct count is updated whenever a correct answer is made.



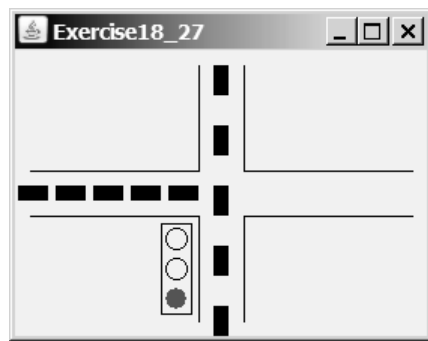
(a) Before a session starts



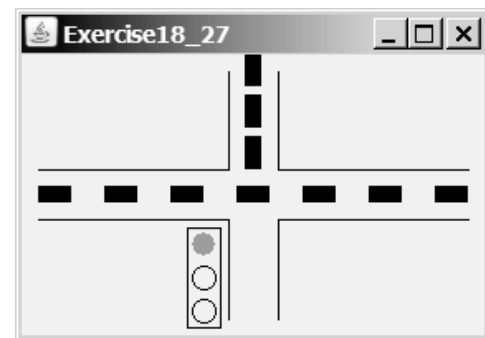
(b) After a session is started

FIGURE 18.29 The applet tests math skills.

18.27* (*Simulation: traffic control*) Exercise 17.3 uses the radio buttons to change the traffic lights. Revise the program that simulates traffic control at an intersection, as shown in Figure 18.30. When the light turns red, the traffic flows



(a) The traffic flows vertically

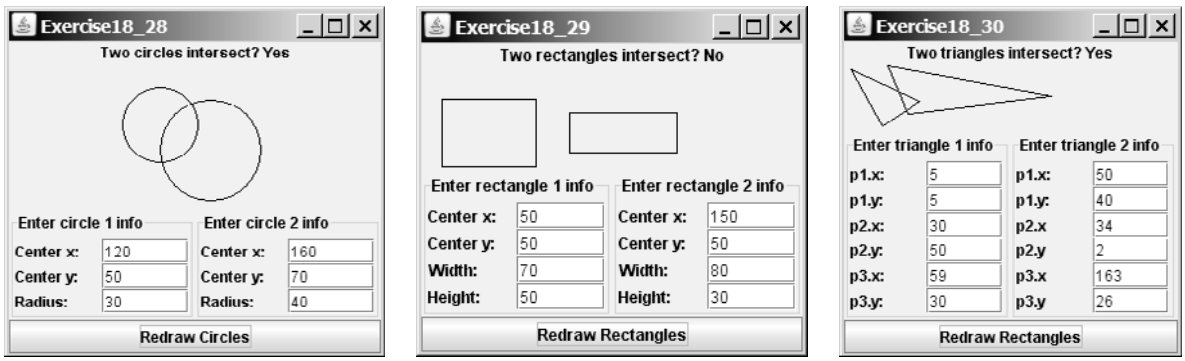


(b) The traffic flows horizontally

FIGURE 18.30 The applet simulates traffic control.

vertically; when the light turns green, the traffic flows horizontally. The light changes automatically every one minute. Before the light changes to red from green, it first changes to yellow for a brief five seconds.

18.28** (*Geometry: two circles intersect?*) The `Circle2D` class was defined in Exercise 10.11. Write an applet that enables the user to specify the location and size of the circles and displays whether the two circles intersect, as shown in Figure 18.31(a).



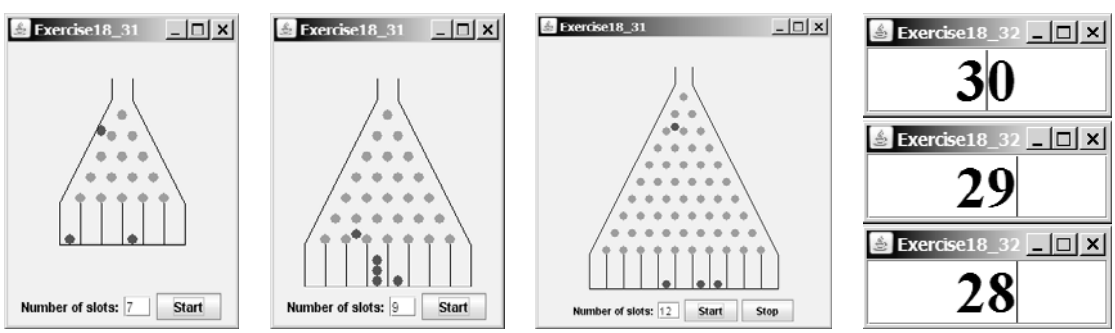
(a) (b) (c)

FIGURE 18.31 Check whether two circles, two rectangles, and two triangles are overlapping.

18.29** (*Geometry: two rectangles intersect?*) The `MyRectangle2D` class was defined in Exercise 10.12. Write an applet that enables the user to specify the location and size of the rectangles and displays whether the two rectangles intersect, as shown in Figure 18.31(b).

18.30** (*Geometry: two triangles intersect?*) The `Triangle2D` class was defined in Exercise 10.13. Write an applet that enables the user to specify the location of the two triangles and displays whether the two triangles intersect, as shown in Figure 18.31(c).

18.31*** (*Game: bean-machine animation*) Write an applet that animates a bean machine introduced in Exercise 16.22. The applet lets you set the number of slots, as shown in Figure 18.32. Click *Start* to start or restart the animation and click *Stop* to stop.



(a) (b) (c) (d)

FIGURE 18.32 (a)–(c) The applet controls a bean-machine animation. (d) The applet counts down the time.

18.32* (*Count-down alarm*) Write an applet that allows the user to enter time in minutes in the text field and press the *Enter* key to count down the minutes, as shown in Figure 18.32(d). The remaining minutes are redisplayed every

one minute. When the minutes are expired, the program starts to play music continuously.

18.33** (*Pattern recognition: consecutive four equal numbers*) Write an applet for Exercise 7.19, as shown in Figure 18.33 (a–b). Let the user enter the numbers in the text fields in a grid of 6 rows and 7 columns. The user can click the *Solve* button to highlight a sequence of four equal numbers, if it exists.



(a) (b) (c)

FIGURE 18.33 (a)–(b) Clicking the *Solve* button to highlight the four consecutive numbers in a row, a column, or a diagonal. (c) The applet enables two players to play the connect-four game.

18.34*** (*Game: connect four*) Exercise 7.20 enables two players to play the connect-four game on the console. Rewrite the program using an applet, as shown in Figure 18.33(c). The applet enables two players to place red and yellow discs in turn. To place a disk, the player needs to click on an available cell. An *available cell* is unoccupied and whose downward neighbor is occupied. The applet flashes the four winning cells if a player wins and reports no winners if all cells are occupied with no winners.

CHAPTER 19

BINARY I/O

Objectives

- To discover how I/O is processed in Java (§19.2).
- To distinguish between text I/O and binary I/O (§19.3).
- To read and write bytes using `FileInputStream` and `FileOutputStream` (§19.4.1).
- To filter data using base classes `FilterInputStream`/`FilterOutputStream` (§19.4.2).
- To read and write primitive values and strings using `DataInputStream`/`DataOutputStream` (§19.4.3).
- To store and restore objects using `ObjectOutputStream` and `ObjectInputStream`, and to understand how objects are serialized and what kind of objects can be serialized (§19.6).
- To implement the `Serializable` interface to make objects serializable (§19.6.1).
- To serialize arrays (§19.6.2).
- To read and write files using the `RandomAccessFile` class (§19.7).



19.1 Introduction

text file
binary file
why binary I/O?

Data stored in a text file are represented in human-readable form. Data stored in a binary file are represented in binary form. You cannot read binary files. They are designed to be read by programs. For example, Java source programs are stored in text files and can be read by a text editor, but Java classes are stored in binary files and are read by the JVM. The advantage of binary files is that they are more efficient to process than text files.

Although it is not technically precise and correct, you can envision a text file as consisting of a sequence of characters and a binary file as consisting of a sequence of bits. For example, the decimal integer `199` is stored as the sequence of three characters, `'1'`, `'9'`, `'9'`, in a text file, and the same integer is stored as a `byte`-type value `C7` in a binary file, because decimal `199` equals hex `C7` ($199 = 12 \times 16^1 + 7$).

text I/O
binary I/O

Java offers many classes for performing file input and output. These can be categorized as *text I/O classes* and *binary I/O classes*. In §9.7, "File Input and Output," you learned how to read/write strings and numeric values from/to a text file using `Scanner` and `PrintWriter`. This section introduces the classes for performing binary I/O.

19.2 How is I/O Handled in Java?

Recall that a `File` object encapsulates the properties of a file or a path but does not contain the methods for reading/writing data from/to a file. In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file. For example, to write text to a file named **temp.txt**, you may create an object using the `PrintWriter` class as follows:

```
PrintWriter output = new PrintWriter("temp.txt");
```

You can now invoke the `print` method from the object to write a string into the file. For example, the following statement writes `"Java 101"` to the file.

```
output.print("Java 101");
```

The next statement closes the file.

```
output.close();
```

There are many I/O classes for various purposes. In general, these can be classified as input classes and output classes. An input class contains the methods to read data, and an output class contains the methods to write data. `PrintWriter` is an example of an output class, and `Scanner` is an example of an input class. The following code creates an input object for the file **temp.txt** and reads data from the file.

```
Scanner input = new Scanner(new File("temp.txt"));
System.out.println(input.nextLine());
```

If **temp.txt** contains `"Java 101"`, `input.nextLine()` returns string `"Java 101"`.

input stream
output stream

Figure 19.1 illustrates Java I/O programming. An input object reads a stream of data from a file, and an output object writes a stream of data to a file. An input object is also called an *input stream* and an output object an *output stream*.

19.3 Text I/O vs. Binary I/O

Computers do not differentiate binary files and text files. All files are stored in binary format, and thus all files are essentially binary files. Text I/O is built upon binary I/O to provide a level of abstraction for character encoding and decoding, as shown in Figure 19.2(a). Encoding and decoding are automatically performed for text I/O. The JVM converts a Unicode to a file-specific

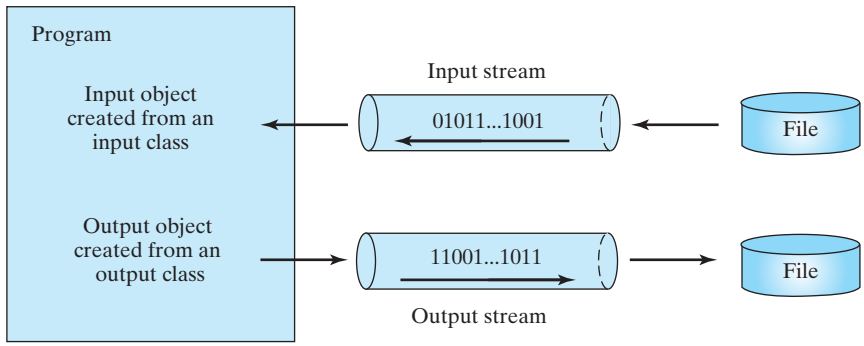


FIGURE 19.1 The program receives data through an input object and sends data through an output object.

encoding when writing a character and converts a file-specific encoding to a Unicode when reading a character. For example, suppose you write string `"199"` using text I/O to a file. Each character is written to the file. Since the Unicode for character `'1'` is `0x0031`, the Unicode `0x0031` is converted to a code that depends on the encoding scheme for the file. (Note that the prefix `0x` denotes a hex number.) In the United States, the default encoding for text files on Windows is ASCII. The ASCII code for character `'1'` is `49` (`0x31` in hex) and for character `'9'` is `57` (`0x39` in hex). So to write the characters `"199"`, three bytes—`0x31`, `0x39`, and `0x39`—are sent to the output, as shown in Figure 19.2(a).



Note

The new version of Java supports supplementary Unicode. For simplicity, however, this book considers only the original Unicode from `0` to `FFFF`.

supplementary Unicode

Binary I/O does not require conversions. If you write a numeric value to a file using binary I/O, the exact value in the memory is copied into the file. For example, a byte-type value `199` is represented as `0xC7` ($199 = 12 \times 16^1 + 7$) in the memory and appears exactly as `0xC7` in the file, as shown in Figure 19.2(b). When you read a byte using binary I/O, one byte value is read from the input.

In general, you should use text input to read a file created by a text editor or a text output program, and use binary input to read a file created by a Java binary output program.

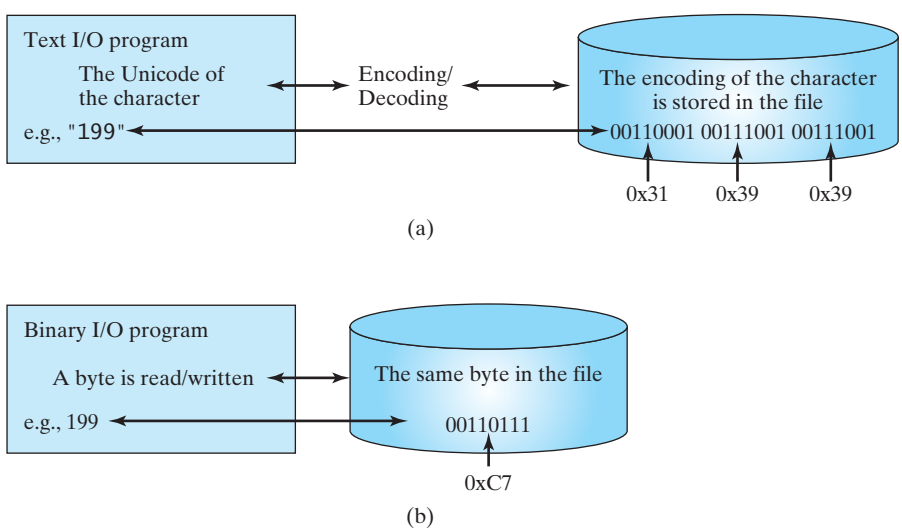


FIGURE 19.2 Text I/O requires encoding and decoding, whereas binary I/O does not.

Binary I/O is more efficient than text I/O, because binary I/O does not require encoding and decoding. Binary files are independent of the encoding scheme on the host machine and thus are portable. Java programs on any machine can read a binary file created by a Java program. This is why Java class files are binary files. Java class files can run on a JVM on any machine.

.txt and .dat



Note

For consistency, this book uses the extension `.txt` to name text files and `.dat` to name binary files.

19.4 Binary I/O Classes

The design of the Java I/O classes is a good example of applying inheritance, where common operations are generalized in superclasses, and subclasses provide specialized operations. Figure 19.3 lists some of the classes for performing binary I/O. `InputStream` is the root for binary input classes, and `OutputStream` is the root for binary output classes. Figures 19.4 and 19.5 list all the methods in `InputStream` and `OutputStream`.

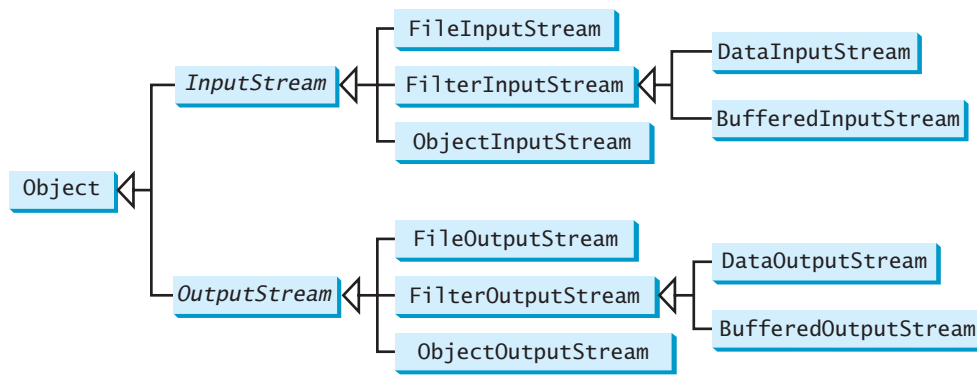


FIGURE 19.3 `InputStream`, `OutputStream`, and their subclasses are for binary I/O.

java.io.InputStream	
+read(): int	Reads the next byte of data from the input stream. The value byte is returned as an `int` value in the range `0` to `255`. If no byte is available because the end of the stream has been reached, the value –1 is returned.
+read(b: byte[]): int	Reads up to `b.length` bytes into array `b` from the input stream and returns the actual number of bytes read. Returns –1 at the end of the stream.
+read(b: byte[], off: int, len: int): int	Reads bytes from the input stream and stores them in `b[off]`, `b[off+1]`, . . ., `b[off+len-1]`. The actual number of bytes read is returned. Returns –1 at the end of the stream.
+available(): int	Returns an estimate of the number of bytes that can be read from the input stream.
+close(): void	Closes this input stream and releases any system resources occupied by it.
+skip(n: long): long	Skips over and discards `n` bytes of data from this input stream. The actual number of bytes skipped is returned.
+markSupported(): boolean	Tests whether this input stream supports the `mark` and `reset` methods.
+mark(readlimit: int): void	Marks the current position in this input stream.
+reset(): void	Repositions this stream to the position at the time the `mark` method was last called on this input stream.

FIGURE 19.4 The abstract `InputStream` class defines the methods for the input stream of bytes.

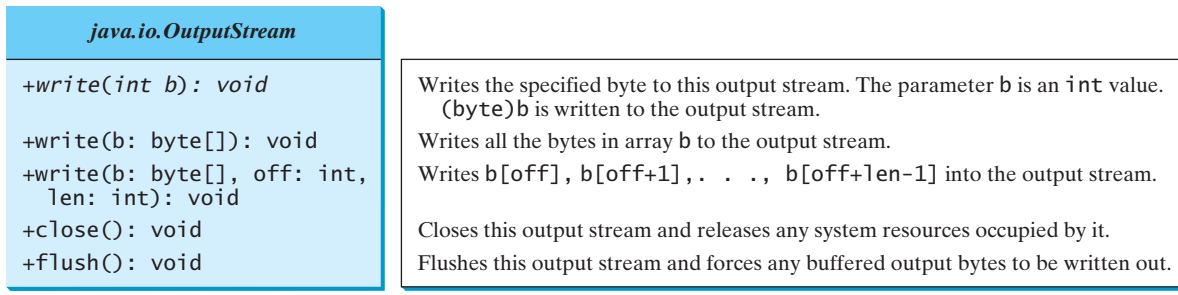

java.io.OutputStream
+write(int b): void
+write(b: byte[]): void
+write(b: byte[], off: int, len: int): void
+close(): void
+flush(): void
Writes the specified byte to this output stream. The parameter b is an int value. (byte)b is written to the output stream.
Writes all the bytes in array b to the output stream.
Writes b[off], b[off+1],. . ., b[off+len-1] into the output stream.
Closes this output stream and releases any system resources occupied by it.
Flushes this output stream and forces any buffered output bytes to be written out.

FIGURE 19.5 The abstract `OutputStream` class defines the methods for the output stream of bytes.



Note

All the methods in the binary I/O classes are declared to throw `java.io.IOException` or a subclass of `java.io.IOException`.

`throws IOException`

19.4.1 `FileInputStream`/`FileOutputStream`

`FileInputStream`/`FileOutputStream` is for reading/writing bytes from/to files. All the methods in these classes are inherited from `InputStream` and `OutputStream`. `FileInputStream`/`FileOutputStream` does not introduce new methods. To construct a `FileInputStream`, use the following constructors, as shown in Figure 19.6:

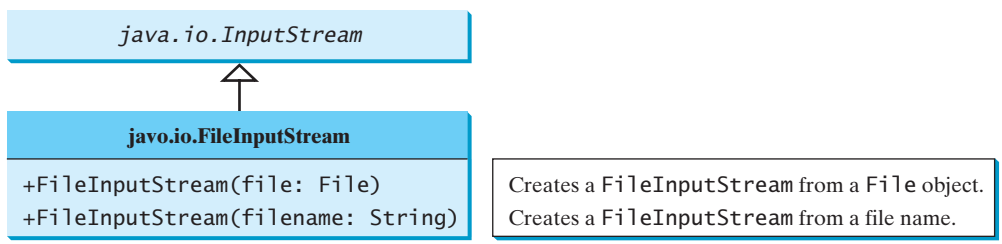


FIGURE 19.6 `FileInputStream` inputs a stream of bytes from a file.

A `java.io.FileNotFoundException` will occur if you attempt to create a `FileInputStream` with a nonexistent file.

`FileNotFoundException`

To construct a `FileOutputStream`, use the constructors shown in Figure 19.7.

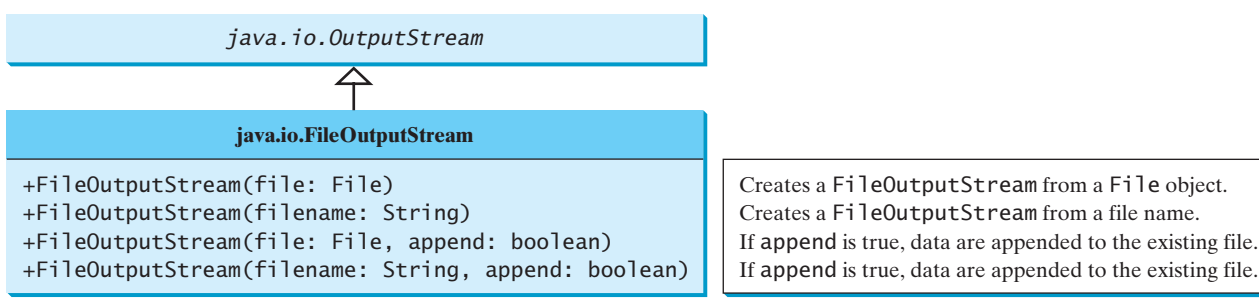

java.io.OutputStream
java.io.FileOutputStream
+FileOutputStream(file: File)
+FileOutputStream(filename: String)
+FileOutputStream(file: File, append: boolean)
+FileOutputStream(filename: String, append: boolean)
Creates a FileOutputStream from a File object.
Creates a FileOutputStream from a file name.
If append is true, data are appended to the existing file.
If append is true, data are appended to the existing file.

FIGURE 19.7 `FileOutputStream` outputs a stream of bytes to a file.

If the file does not exist, a new file will be created. If the file already exists, the first two constructors will delete the current content of the file. To retain the current content and append new data into the file, use the last two constructors by passing `true` to the `append` parameter.

IOException

Almost all the methods in the I/O classes throw `java.io.IOException`. Therefore you have to declare `java.io.IOException` to throw in the method or place the code in a `try-catch` block, as shown below:

Declaring exception in the method

```
public static void main(String[] args)
    throws IOException {
  // Perform I/O operations
}
```

Using try-catch block

```
public static void main(String[] args) {
  try {
    // Perform I/O operations
  }
  catch (IOException ex) {
    ex.printStackTrace();
  }
}
```

Listing 19.1 uses binary I/O to write ten byte values from `1` to `10` to a file named **temp.dat** and reads them back from the file.

LISTING 19.1 TestFileStream.java

import

output stream

output

input stream

input

```
1  import java.io.*;
2
3  public class TestFileStream {
4    public static void main(String[] args) throws IOException {
5      // Create an output stream to the file
6      FileOutputStream output = new FileOutputStream("temp.dat");
7
8      // Output values to the file
9      for (int i = 1; i <= 10; i++)
10       output.write(i);
11
12     // Close the output stream
13     output.close();
14
15     // Create an input stream for the file
16     FileInputStream input = new FileInputStream("temp.dat");
17
18     // Read values from the file
19     int value;
20     while ((value = input.read()) != -1)
21       System.out.print(value + " ");
22
23     // Close the output stream
24     input.close();
25   }
26 }
```



```
1 2 3 4 5 6 7 8 9 10
```

A `FileOutputStream` is created for file **temp.dat** in line 6. The `for` loop writes ten byte values into the file (lines 9–10). Invoking `write(i)` is the same as invoking `write((byte)i)`. Line 13 closes the output stream. Line 16 creates a `FileInputStream` for file **temp.dat**. Values are read from the file and displayed on the console in lines 19–21. The expression `((value = input.read()) != -1)` (line 20) reads a byte from `input.read()`, assigns it to `value`, and checks whether it is `-1`. The input value of `-1` signifies the end of a file.

end of a file

The file **temp.dat** created in this example is a binary file. It can be read from a Java program but not from a text editor, as shown in Figure 19.8.

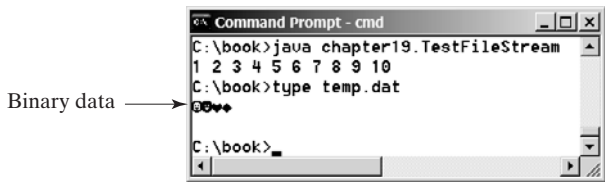


FIGURE 19.8 A binary file cannot be displayed in text mode.



Tip

When a stream is no longer needed, always close it using the `close()` method. Not closing streams may cause data corruption in the output file, or other programming errors.

close stream



Note

The root directory for the file is the classpath directory. For the example in this book, the root directory is **c:\book**. So the file **temp.dat** is located at **c:\book**. If you wish to place **temp.dat** in a specific directory, replace line 8 by

where is the file?

```
FileOutputStream output =
  new FileOutputStream("directory/temp.dat");
```



Note

An instance of `FileInputStream` can be used as an argument to construct a `Scanner`, and an instance of `FileOutputStream` can be used as an argument to construct a `PrintWriter`. You can create a `PrintWriter` to append text into a file using

appending to text file

```
new PrintWriter(new FileOutputStream("temp.txt", true));
```

If **temp.txt** does not exist, it is created. If **temp.txt** already exists, new data are appended to the file.

19.4.2 FilterInputStream/FilterOutputStream

Filter streams are streams that filter bytes for some purpose. The basic byte input stream provides a `read` method that can be used only for reading bytes. If you want to read integers, doubles, or strings, you need a filter class to wrap the byte input stream. Using a filter class enables you to read integers, doubles, and strings instead of bytes and characters. `FilterInputStream` and `FilterOutputStream` are the base classes for filtering data. When you need to process primitive numeric types, use `DataInputStream` and `DataOutputStream` to filter bytes.

19.4.3 DataInputStream/DataOutputStream

`DataInputStream` reads bytes from the stream and converts them into appropriate primitive type values or strings. `DataOutputStream` converts primitive type values or strings into bytes and outputs the bytes to the stream.

`DataInputStream` extends `FilterInputStream` and implements the `DataInput` interface, as shown in Figure 19.9. `DataOutputStream` extends `FilterOutputStream` and implements the `DataOutput` interface, as shown in Figure 19.10.

`DataInputStream` implements the methods defined in the `DataInput` interface to read primitive data type values and strings. `DataOutputStream` implements the methods defined in the `DataOutput` interface to write primitive data type values and strings. Primitive values

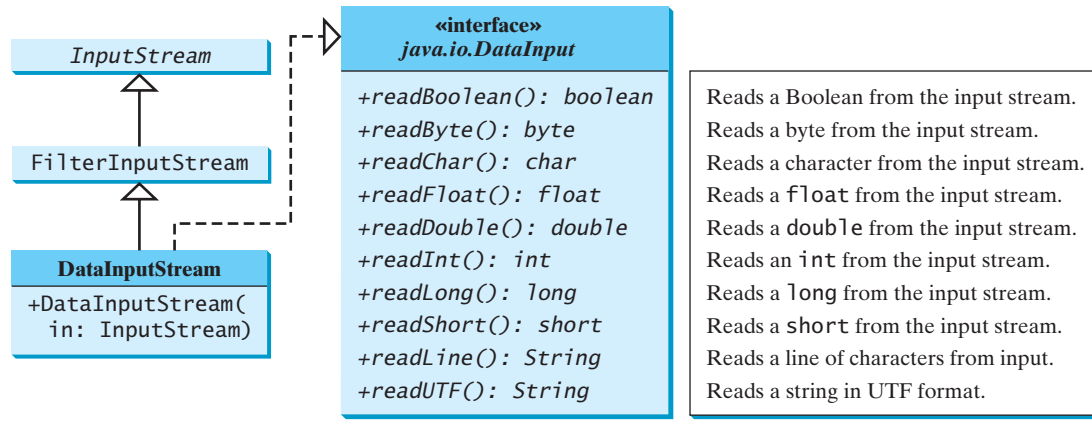


FIGURE 19.9 `DataInputStream` filters an input stream of bytes into primitive data type values and strings.

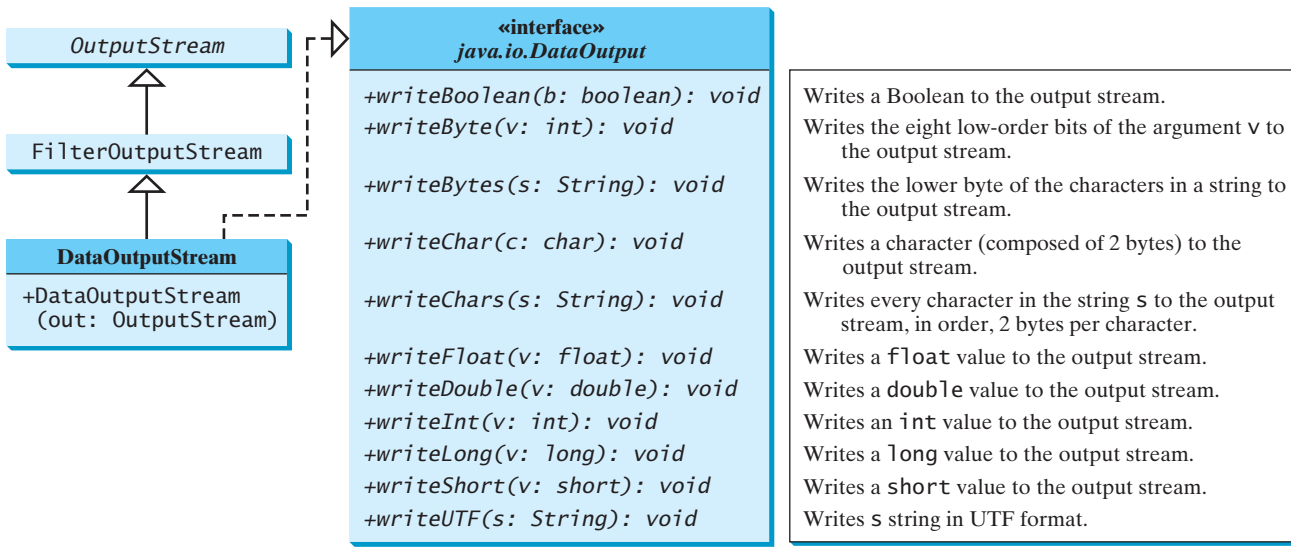

OutputStream
FilterOutputStream
DataOutputStream
+DataOutputStream
(out: OutputStream)
«interface»
java.io.DataOutput
+writeBoolean(b: boolean): void
+writeByte(v: int): void
+writeBytes(s: String): void
+writeChar(c: char): void
+writeChars(s: String): void
+writeFloat(v: float): void
+writeDouble(v: double): void
+writeInt(v: int): void
+writeLong(v: long): void
+writeShort(v: short): void
+writeUTF(s: String): void
Writes a Boolean to the output stream.
Writes the eight low-order bits of the argument v to the output stream.
Writes the lower byte of the characters in a string to the output stream.
Writes a character (composed of 2 bytes) to the output stream.
Writes every character in the string s to the output stream, in order, 2 bytes per character.
Writes a float value to the output stream.
Writes a double value to the output stream.
Writes an int value to the output stream.
Writes a long value to the output stream.
Writes a short value to the output stream.
Writes s string in UTF format.

FIGURE 19.10 `DataOutputStream` enables you to write primitive data type values and strings into an output stream.

are copied from memory to the output without any conversions. Characters in a string may be written in several ways, as discussed in the next section.

Characters and Strings in Binary I/O

A Unicode consists of two bytes. The `writeChar(char c)` method writes the Unicode of character `c` to the output. The `writeChars(String s)` method writes the Unicode for each character in the string `s` to the output. The `writeBytes(String s)` method writes the lower byte of the Unicode for each character in the string `s` to the output. The high byte of the Unicode is discarded. The `writeBytes` method is suitable for strings that consist of ASCII characters, since an ASCII code is stored only in the lower byte of a Unicode. If a string consists of non-ASCII characters, you have to use the `writeChars` method to write the string.

The `writeUTF(String s)` method writes two bytes of length information to the output stream, followed by the modified UTF-8 representation of every character in the string `s`. UTF-8 is a coding scheme that allows systems to operate with both ASCII and Unicode. Most operating systems use ASCII. Java uses Unicode. The ASCII character set is a subset of the Unicode character set. Since most applications need only the ASCII character set, it is a waste to represent an 8-bit ASCII character as a 16-bit Unicode character. The modified UTF-8 scheme stores a character using one, two, or three bytes. Characters are coded in one byte if

their code is less than or equal to `0x7F`, in two bytes if their code is greater than `0x7F` and less than or equal to `0x7FF`, or in three bytes if their code is greater than `0x7FF`.

The initial bits of a UTF-8 character indicate whether a character is stored in one byte, two bytes, or three bytes. If the first bit is `0`, it is a one-byte character. If the first bits are `110`, it is the first byte of a two-byte sequence. If the first bits are `1110`, it is the first byte of a three-byte sequence. The information that indicates the number of characters in a string is stored in the first two bytes preceding the UTF-8 characters. For example, `writeUTF("ABCDEF")` actually writes eight bytes (i.e., `00 06 41 42 43 44 45 46`) to the file, because the first two bytes store the number of characters in the string.

UTF-8 scheme

The `writeUTF(String s)` method converts a string into a series of bytes in the UTF-8 format and writes them into a binary stream. The `readUTF()` method reads a string that has been written using the `writeUTF` method.

The UTF-8 format has the advantage of saving a byte for each ASCII character, because a Unicode character takes up two bytes and an ASCII character in UTF-8 only one byte. If most of the characters in a long string are regular ASCII characters, using UTF-8 is efficient.

Using `DataInputStream`/`DataOutputStream`

Data streams are used as wrappers on existing input, and output streams to filter data in the original stream. They are created using the following constructors (see Figures 19.9 and 19.10):

```
public DataInputStream(InputStream instream)
public DataOutputStream(OutputStream outstream)
```

The statements given below create data streams. The first statement creates an input stream for file **in.dat**; the second statement creates an output stream for file **out.dat**.

```
DataInputStream input =
  new DataInputStream(new FileInputStream("in.dat"));
DataOutputStream output =
  new DataOutputStream(new FileOutputStream("out.dat"));
```

Listing 19.2 writes student names and scores to a file named **temp.dat** and reads the data back from the file.

LISTING 19.2 TestDataStream.java

```
1  import java.io.*;
2
3  public class TestDataStream {
4    public static void main(String[] args) throws IOException {
5      // Create an output stream for file temp.dat
6      DataOutputStream output =                                    output stream
7        new DataOutputStream(new FileOutputStream("temp.dat"));
8
9      // Write student test scores to the file
10     output.writeUTF("John");                                     output
11     output.writeDouble(85.5);
12     output.writeUTF("Jim");
13     output.writeDouble(185.5);
14     output.writeUTF("George");
15     output.writeDouble(105.25);
16
17     // Close output stream
18     output.close();                                              close stream
19
20     // Create an input stream for file temp.dat
21     DataInputStream input =                                      input stream
22       new DataInputStream(new FileInputStream("temp.dat"));
23
```

input

```
24     // Read student test scores from the file
25     System.out.println(input.readUTF() + " " + input.readDouble());
26     System.out.println(input.readUTF() + " " + input.readDouble());
27     System.out.println(input.readUTF() + " " + input.readDouble());
28   }
29 }
```



```
John 85.5
Jim 185.5
George 105.25
```

A `DataOutputStream` is created for file **temp.dat** in lines 6–7. Student names and scores are written to the file in lines 10–15. Line 18 closes the output stream. A `DataInputStream` is created for the same file in lines 21–22. Student names and scores are read back from the file and displayed on the console in lines 25–27.

`DataInputStream` and `DataOutputStream` read and write Java primitive type values and strings in a machine-independent fashion, thereby enabling you to write a data file on one machine and read it on another machine that has a different operating system or file structure. An application uses a data output stream to write data that can later be read by a program using a data input stream.



Caution

You have to read data in the same order and format in which they are stored. For example, since names are written in UTF-8 using `writeUTF`, you must read names using `readUTF`.

Detecting End of File

`EOFException`

If you keep reading data at the end of an `InputStream`, an `EOFException` will occur. This exception may be used to detect the end of file, as shown in Listing 19.3.

LISTING 19.3 DetectEndOfFile.java

output stream

output

close stream

input stream

input

`EOFException`

```
1  import java.io.*;
2
3  public class DetectEndOfFile {
4    public static void main(String[] args) {
5      try {
6        DataOutputStream output = new DataOutputStream
7          (new FileOutputStream("test.dat"));
8        output.writeDouble(4.5);
9        output.writeDouble(43.25);
10       output.writeDouble(3.2);
11       output.close();
12
13       DataInputStream input = new DataInputStream
14         (new FileInputStream("test.dat"));
15       while (true) {
16         System.out.println(input.readDouble());
17       }
18     }
19     catch (EOFException ex) {
20       System.out.println("All data read");
21     }
22     catch (IOException ex) {
23       ex.printStackTrace();
24     }
25   }
26 }
```

```
4.5
43.25
3.2
All data read
```



The program writes three double values to the file using `DataOutputStream` (lines 6–10), and reads the data using `DataInputStream` (lines 13–17). When reading past the end of file, an `EOFException` is thrown. The exception is caught in line 19.

19.4.4 `BufferedInputStream`/`BufferedOutputStream`

`BufferedInputStream`/`BufferedOutputStream` can be used to speed up input and output by reducing the number of reads and writes. `BufferedInputStream`/`BufferedOutputStream` does not contain new methods. All the methods in `BufferedInputStream`/`BufferedOutputStream` are inherited from the `InputStream`/`OutputStream` classes. `BufferedInputStream`/`BufferedOutputStream` adds a buffer in the stream for storing bytes for efficient processing.

You may wrap a `BufferedInputStream`/`BufferedOutputStream` on any `InputStream`/`OutputStream` using the constructors shown in Figures 19.11 and 19.12.

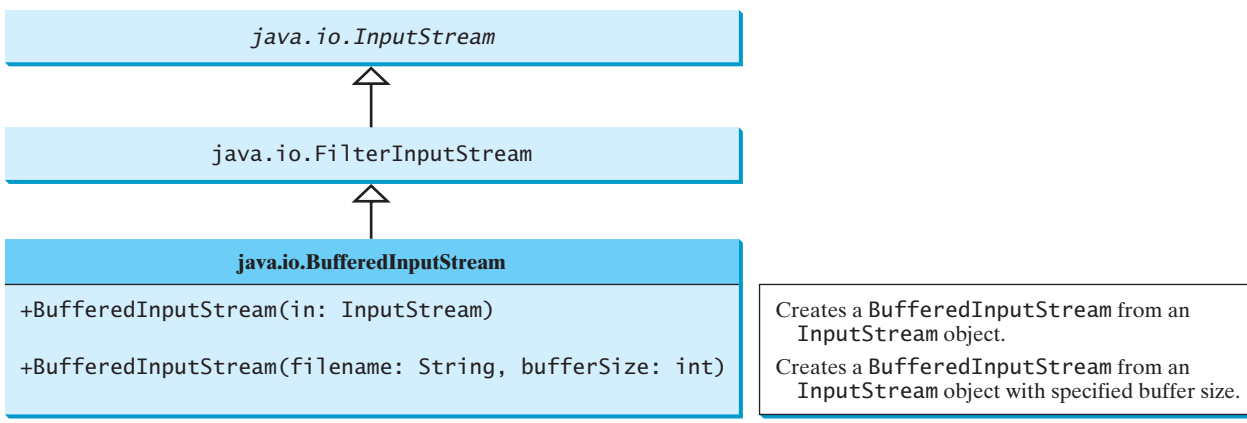


FIGURE 19.11 `BufferedInputStream` buffers input stream.

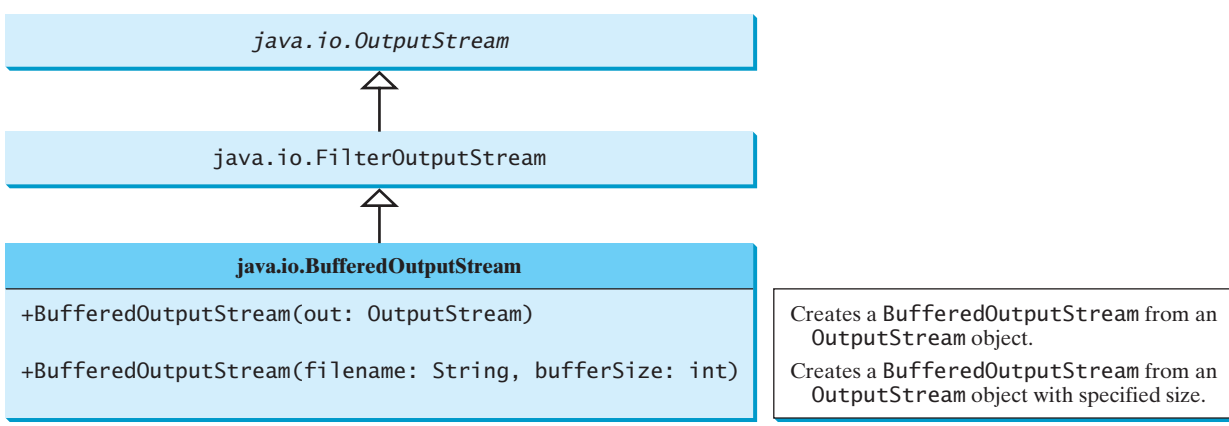


FIGURE 19.12 `BufferedOutputStream` buffers output stream.

If no buffer size is specified, the default size is `512` bytes. A buffered input stream reads as many data as possible into its buffer in a single read call. By contrast, a buffered output stream calls the write method only when its buffer fills up or when the `flush()` method is called.

You can improve the performance of the `TestDataStream` program in the preceding example by adding buffers in the stream in lines 6–7 and 13–14 as follows:

```
DataOutputStream output = new DataOutputStream(
  new BufferedOutputStream (new FileOutputStream("temp.dat")));
DataInputStream input = new DataInputStream(
  new BufferedInputStream (new FileInputStream("temp.dat")));
```



Tip

You should always use buffered IO to speed up input and output. For small files, you may not notice performance improvements. However, for large files—over 100 MB—you will see substantial improvements using buffered IO.

19.5 Problem: Copying Files



Video Note
Copy file

This section develops a program that copies files. The user needs to provide a source file and a target file as command-line arguments using the following command:

```
java Copy source target
```

The program copies a source file to a target file and displays the number of bytes in the file. If the source does not exist, the user is told that the file has not been found. If the target file already exists, the user is told that the file exists. A sample run of the program is shown in Figure 19.13.

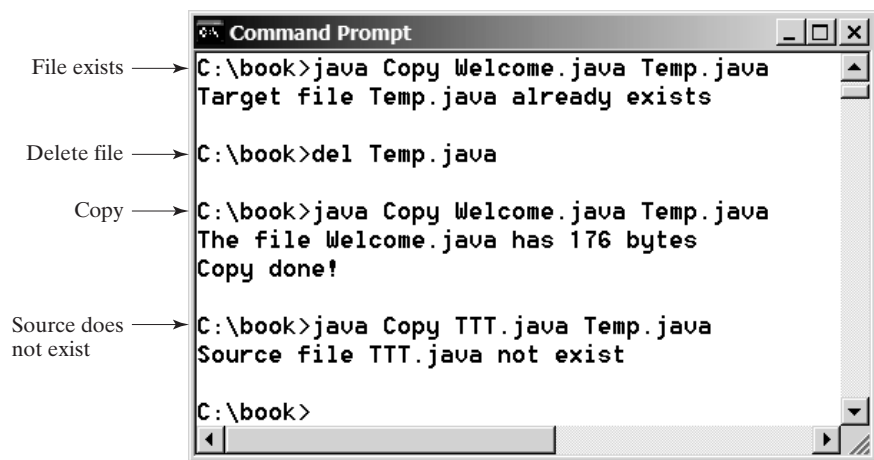


FIGURE 19.13 The program copies a file.

To copy the contents from a source to a target file, it is appropriate to use a binary input stream to read bytes from the source file and a binary output stream to send bytes to the target file, regardless of the contents of the file. The source file and the target file are specified from the command line. Create an `InputFileStream` for the source file and an `OutputFileStream` for the target file. Use the `read()` method to read a byte from the input stream, and then use the `write(b)` method to write the byte to the output stream. Use `BufferedInputStream` and `BufferedOutputStream` to improve the performance. Listing 19.4 gives the solution to the problem.

LISTING 19.4 Copy.java

```
1 import java.io.*;
2
3 public class Copy {
4   /** Main method
5      @param args[0] for sourcefile
6      @param args[1] for target file
7    */
8   public static void main(String[] args) throws IOException {
9     // Check command-line parameter usage
10    if (args.length != 2) {                                          check usage
11      System.out.println(
12        "Usage: java Copy sourceFile targetfile");
13      System.exit(0);
14    }
15
16    // Check whether source file exists
17    File sourceFile = new File(args[0]);                             source file
18    if (!sourceFile.exists()) {
19       System.out.println("Source file " + args[0] + " not exist");
20       System.exit(0);
21    }
22
23    // Check whether target file exists
24    File targetFile = new File(args[1]);                             target file
25    if (targetFile.exists()) {
26      System.out.println("Target file " + args[1] + " already
27        exists");
28      System.exit(0);
29    }
30
31    // Create an input stream
32    BufferedInputStream input =                                      input stream
33      new BufferedInputStream(new FileInputStream(sourceFile));
34
35    // Create an output stream
36    BufferedOutputStream output =                                    output stream
37      new BufferedOutputStream(new FileOutputStream(targetFile));
38
39    // Continuously read a byte from input and write it to output
40    int r; int numberOfBytesCopied = 0;
41    while ((r = input.read()) != -1) {                               read
42      output.write((byte)r);                                         write
43      numberOfBytesCopied++;
44    }
45
46    // Close streams
47    input.close();                                                   close stream
48    output.close();
49
50    // Display the file size
51    System.out.println(numberOfBytesCopied + " bytes copied");
52  }
53 }
```

The program first checks whether the user has passed two required arguments from the command line in lines 10–14.

The program uses the `File` class to check whether the source file and target file exist. If the source file does not exist (lines 18–21) or if the target file already exists, exit the program.

An input stream is created using `BufferedInputStream` wrapped on `FileInputStream` in lines 32–33, and an output stream is created using `BufferedOutputStream` wrapped on `FileOutputStream` in lines 36–37.

The expression `((r = input.read()) != -1)` (line 41) reads a byte from `input.read()`, assigns it to `r`, and checks whether it is `-1`. The input value of `-1` signifies the end of a file. The program continuously reads bytes from the input stream and sends them to the output stream until all of the bytes have been read.

19.6 Object I/O



Video Note
Object I/O

`DataInputStream`/`DataOutputStream` enables you to perform I/O for primitive type values and strings. `ObjectInputStream`/`ObjectOutputStream` enables you to perform I/O for objects in addition to primitive type values and strings. Since `ObjectInputStream`/`ObjectOutputStream` contains all the functions of `DataInputStream`/`DataOutputStream`, you can replace `DataInputStream`/`DataOutputStream` completely with `ObjectInputStream`/`ObjectOutputStream`.

`ObjectInputStream` extends `InputStream` and implements `ObjectInput` and `ObjectStreamConstants`, as shown in Figure 19.14. `ObjectInput` is a subinterface of `DataInput`. `DataInput` is shown in Figure 19.9. `ObjectStreamConstants` contains the constants to support `ObjectInputStream`/`ObjectOutputStream`.

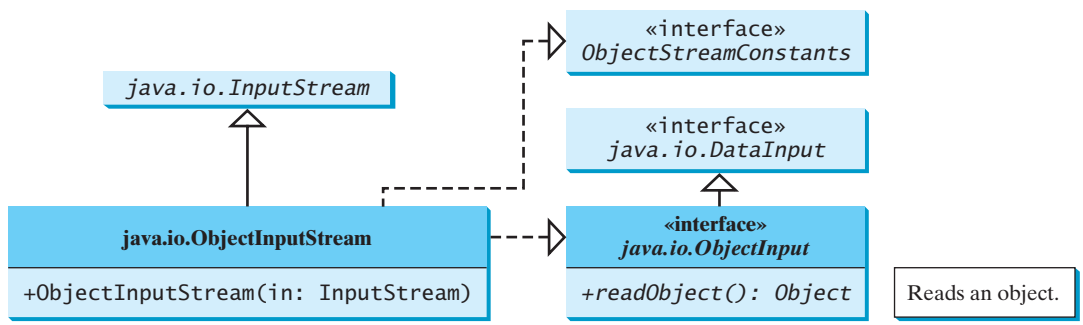


FIGURE 19.14 `ObjectInputStream` can read objects, primitive type values, and strings.

`ObjectOutputStream` extends `OutputStream` and implements `ObjectOutput` and `ObjectStreamConstants`, as shown in Figure 19.15. `ObjectOutput` is a subinterface of `DataOutput`. `DataOutput` is shown in Figure 19.10.

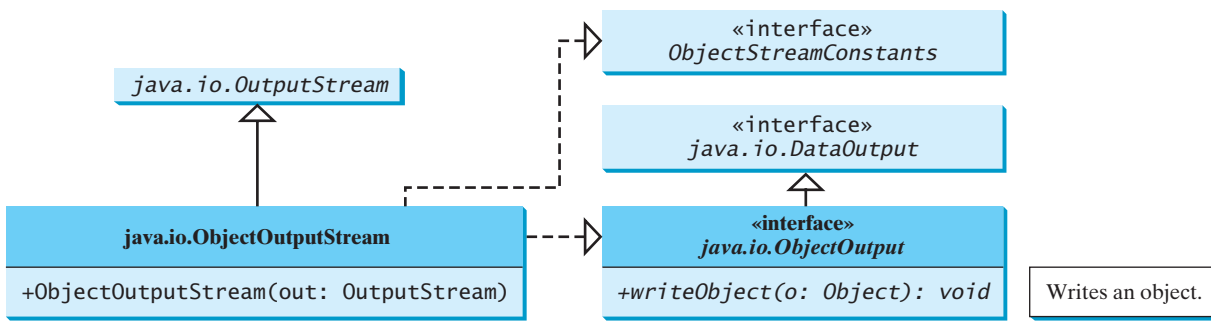


FIGURE 19.15 `ObjectOutputStream` can write objects, primitive type values, and strings.

You may wrap an `ObjectInputStream`/`ObjectOutputStream` on any `InputStream`/`OutputStream` using the following constructors:

```
// Create an ObjectInputStream
public ObjectInputStream(InputStream in)

// Create an ObjectOutputStream
public ObjectOutputStream(OutputStream out)
```

Listing 19.5 writes student names, scores, and current date to a file named **object.dat**.

LISTING 19.5 TestObjectOutputStream.java

```
1 import java.io.*;
2
3 public class TestObjectOutputStream {
4   public static void main(String[] args) throws IOException {
5     // Create an output stream for file object.dat
6     ObjectOutputStream output =                                    output stream
7       new ObjectOutputStream(new FileOutputStream("object.dat"));
8
9     // Write a string, double value, and object to the file
10    output.writeUTF("John");                                       output
11    output.writeDouble(85.5);
12    output.writeObject(new java.util.Date());
13
14    // Close output stream
15    output.close();
16  }
17 }
```

An `ObjectOutputStream` is created to write data into file **object.dat** in lines 6–7. A string, a double value, and an object are written to the file in lines 10–12. To improve performance, you may add a buffer in the stream using the following statement to replace lines 6–7:

```
ObjectOutputStream output = new ObjectOutputStream(
  new BufferedOutputStream(new FileOutputStream("object.dat")));
```

Multiple objects or primitives can be written to the stream. The objects must be read back from the corresponding `ObjectInputStream` with the same types and in the same order as they were written. Java's safe casting should be used to get the desired type. Listing 19.6 reads data back from **object.dat**.

LISTING 19.6 TestObjectInputStream.java

```
1 import java.io.*;
2
3 public class TestObjectInputStream {
4   public static void main(String[] args)
5     throws ClassNotFoundException, IOException {
6     // Create an input stream for file object.dat
7     ObjectInputStream input =                                      input stream
8       new ObjectInputStream(new FileInputStream("object.dat"));
9
10    // Write a string, double value, and object to the file
11    String name = input.readUTF();                                 input
12    double score = input.readDouble();
13    java.util.Date date = (java.util.Date)(input.readObject());
14    System.out.println(name + " " + score + " " + date);
15
```

```
16     // Close output stream
17     input.close();
18   }
19 }
```



```
John 85.5 Mon Jun 26 17:17:29 EDT 2006
```

ClassNotFoundException

The `readObject()` method may throw `java.lang.ClassNotFoundException`. The reason is that when the JVM restores an object, it first loads the class for the object if the class has not been loaded. Since `ClassNotFoundException` is a checked exception, the `main` method declares to throw it in line 5. An `ObjectInputStream` is created to read input from **object.dat** in lines 7–8. You have to read the data from the file in the same order and format as they were written to the file. A string, a double value, and an object are read in lines 11–13. Since `readObject()` returns an `Object`, it is cast into `Date` and assigned to a `Date` variable in line 13.

19.6.1 The `Serializable` Interface

serializable

Not every object can be written to an output stream. Objects that can be so written are said to be *serializable*. A serializable object is an instance of the `java.io.Serializable` interface, so the object's class must implement `Serializable`.

The `Serializable` interface is a marker interface. Since it has no methods, you don't need to add additional code in your class that implements `Serializable`. Implementing this interface enables the Java serialization mechanism to automate the process of storing objects and arrays.

To appreciate this automation feature, consider what you otherwise need to do in order to store an object. Suppose you want to store a `JButton` object. To do this you need to store all the current values of the properties (e.g., color, font, text, alignment) in the object. Since `JButton` is a subclass of `AbstractButton`, the property values of `AbstractButton` have to be stored as well as the properties of all the superclasses of `AbstractButton`. If a property is of an object type (e.g., `background` of the `Color` type), storing it requires storing all the property values inside this object. As you can see, this is a very tedious process. Fortunately, you don't have to go through it manually. Java provides a built-in mechanism to automate the process of writing objects. This process is referred to as *object serialization*, which is implemented in `ObjectOutputStream`. In contrast, the process of reading objects is referred to as *object deserialization*, which is implemented in `ObjectInputStream`.

serialization

deserialization

Many classes in the Java API implement `Serializable`. The utility classes, such as `java.util.Date`, and all the Swing GUI component classes implement `Serializable`. Attempting to store an object that does not support the `Serializable` interface would cause a `NotSerializableException`.

NotSerializable-Exception

When a serializable object is stored, the class of the object is encoded; this includes the class name and the signature of the class, the values of the object's instance variables, and the closure of any other objects referenced from the initial object. The values of the object's static variables are not stored.



Note

nonserializable fields

If an object is an instance of `Serializable` but contains nonserializable instance data fields, can it be serialized? The answer is no. To enable the object to be serialized, mark these data fields with the `transient` keyword to tell the JVM to ignore them when writing the object to an object stream. Consider the following class:

transient

```
public class Foo implements java.io.Serializable {
  private int v1;
  private static double v2;
  private transient A v3 = new A();
}

class A { } // A is not serializable
```

When an object of the `Foo` class is serialized, only variable `v1` is serialized. Variable `v2` is not serialized because it is a static variable, and variable `v3` is not serialized because it is marked `transient`. If `v3` were not marked `transient`, a `java.io.NotSerializableException` would occur.



Note

duplicate objects

If an object is written to an object stream more than once, will it be stored in multiple copies? No, it will not. When an object is written for the first time, a serial number is created for it. The JVM writes the complete content of the object along with the serial number into the object stream. After the first time, only the serial number is stored if the same object is written again. When the objects are read back, their references are the same, since only one object is actually created in the memory.

19.6.2 Serializing Arrays

An array is serializable if all its elements are serializable. An entire array can be saved using `writeObject` into a file and later can be restored using `readObject`. Listing 19.7 stores an array of five `int` values and an array of three strings and reads them back to display on the console.

LISTING 19.7 TestObjectStreamForArray.java

```
1  import java.io.*;
2
3  public class TestObjectStreamForArray {
4    public static void main(String[] args)
5        throws ClassNotFoundException, IOException {
6      int[] numbers = {1, 2, 3, 4, 5};
7      String[] strings = {"John", "Jim", "Jake"};
8
9      // Create an output stream for file array.dat
10     ObjectOutputStream output =                                  output stream
11       new ObjectOutputStream(new FileOutputStream
12         ("array.dat", true));
13
14     // Write arrays to the object output stream                  store array
15     output.writeObject(numbers);
16     output.writeObject(strings);
17
18     // Close the stream
19     output.close();
20
21     // Create an input stream for file array.dat
22     ObjectInputStream input =                                    input stream
23       new ObjectInputStream(new FileInputStream("array.dat"));
24
25     int[] newNumbers = (int[])(input.readObject());              restore array
26     String[] newStrings = (String[])(input.readObject());
27
28     // Display arrays
29     for (int i = 0; i < newNumbers.length; i++)
30       System.out.print(newNumbers[i] + " ");
31     System.out.println();
32
33     for (int i = 0; i < newStrings.length; i++)
34       System.out.print(newStrings[i] + " ");
35   }
36 }
```



```
1 2 3 4 5
John Jim Jake
```

Lines 15–16 write two arrays into file **array.dat**. Lines 25–26 read three arrays back in the same order they were written. Since `readObject()` returns `Object`, casting is used to cast the objects into `int[]` and `String[]`.

19.7 Random-Access Files

read-only

write-only

sequential

All of the streams you have used so far are known as *read-only* or *write-only* streams. The external files of these streams are *sequential* files that cannot be updated without creating a new file. It is often necessary to modify files or to insert new records into files. Java provides the `RandomAccessFile` class to allow a file to be read from and written to at random locations.

The `RandomAccessFile` class implements the `DataInput` and `DataOutput` interfaces, as shown in Figure 19.16. The `DataInput` interface shown in Figure 19.9 defines the methods (e.g., `readInt`, `readDouble`, `readChar`, `readBoolean`, `readUTF`) for reading primitive type values and strings, and the `DataOutput` interface shown in Figure 19.10 defines the methods (e.g., `writeInt`, `writeDouble`, `writeChar`, `writeBoolean`, `writeUTF`) for writing primitive type values and strings.

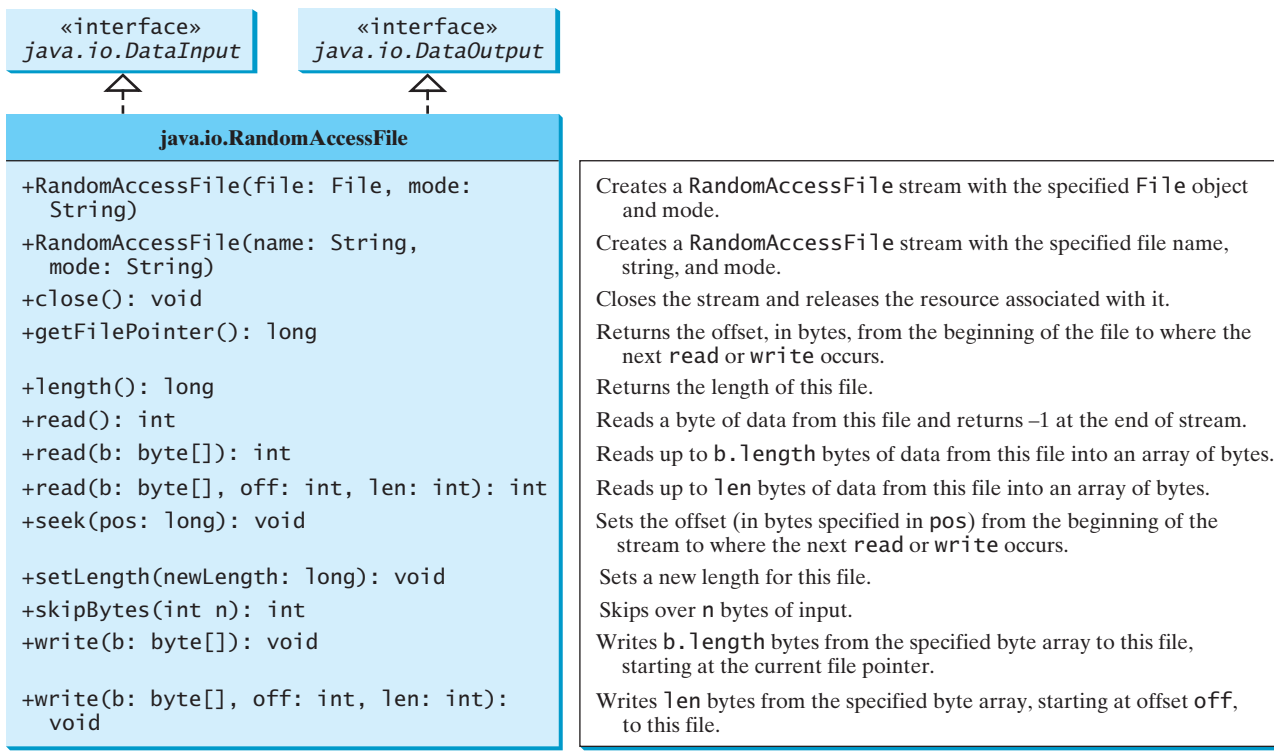

«interface»
java.io.DataInput
«interface»
java.io.DataOutput
java.io.RandomAccessFile
+RandomAccessFile(file: File, mode: String)
+RandomAccessFile(name: String, mode: String)
+close(): void
+getFilePointer(): long
+length(): long
+read(): int
+read(b: byte[]): int
+read(b: byte[], off: int, len: int): int
+seek(pos: long): void
+setLength(newLength: long): void
+skipBytes(int n): int
+write(b: byte[]): void
+write(b: byte[], off: int, len: int): void
Creates a RandomAccessFile stream with the specified File object and mode.
Creates a RandomAccessFile stream with the specified file name, string, and mode.
Closes the stream and releases the resource associated with it.
Returns the offset, in bytes, from the beginning of the file to where the next read or write occurs.
Returns the length of this file.
Reads a byte of data from this file and returns –1 at the end of stream.
Reads up to b.length bytes of data from this file into an array of bytes.
Reads up to len bytes of data from this file into an array of bytes.
Sets the offset (in bytes specified in pos) from the beginning of the stream to where the next read or write occurs.
Sets a new length for this file.
Skips over n bytes of input.
Writes b.length bytes from the specified byte array to this file, starting at the current file pointer.
Writes len bytes from the specified byte array, starting at offset off, to this file.

FIGURE 19.16 `RandomAccessFile` implements the `DataInput` and `DataOutput` interfaces with additional methods to support random access.

When creating a `RandomAccessFile`, you can specify one of two modes ("`r`" or "`rw`"). Mode "`r`" means that the stream is read-only, and mode "`rw`" indicates that the stream allows

both read and write. For example, the following statement creates a new stream, `raf`, that allows the program to read from and write to the file **test.dat**:

```
RandomAccessFile raf = new RandomAccessFile("test.dat", "rw");
```

If **test.dat** already exists, `raf` is created to access it; if **test.dat** does not exist, a new file named **test.dat** is created, and `raf` is created to access the new file. The method `raf.length()` returns the number of bytes in **test.dat** at any given time. If you append new data into the file, `raf.length()` increases.



Tip

If the file is not intended to be modified, open it with the `"r"` mode. This prevents unintentional modification of the file.

A random-access file consists of a sequence of bytes. A special marker called a *file pointer* is positioned at one of these bytes. A read or write operation takes place at the location of the file pointer. When a file is opened, the file pointer is set at the beginning of the file. When you read or write data to the file, the file pointer moves forward to the next data item. For example, if you read an `int` value using `readInt()`, the JVM reads `4` bytes from the file pointer, and now the file pointer is `4` bytes ahead of the previous location, as shown in Figure 19.17.

file pointer

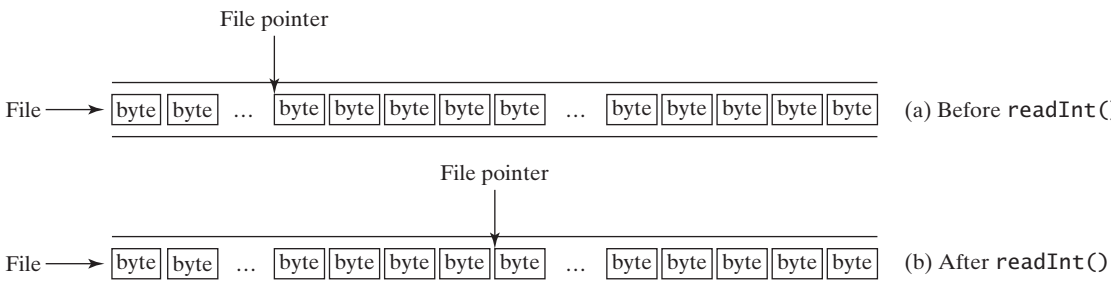


FIGURE 19.17 After an `int` value is read, the file pointer is moved 4 bytes ahead.

For a `RandomAccessFile raf`, you can use the `raf.seek(position)` method to move the file pointer to a specified position. `raf.seek(0)` moves it to the beginning of the file, and `raf.seek(raf.length())` moves it to the end of the file. Listing 19.8 demonstrates `RandomAccessFile`. A large case study of using `RandomAccessFile` to organize an address book is given in Supplement VII.B.

LISTING 19.8 TestRandomAccessFile.java

```
1  import java.io.*;
2
3  public class TestRandomAccessFile {
4    public static void main(String[] args) throws IOException {
5      // Create a random-access file
6      RandomAccessFile inout = new RandomAccessFile("inout.dat", "rw");
7
8      // Clear the file to destroy the old contents, if any
9      inout.setLength(0);
10
11     // Write new integers to the file
12     for (int i = 0; i < 200; i++)
13       inout.writeInt(i);
14
15     // Display the current length of the file
16     System.out.println("Current file length is " + inout.length());
17
```

RandomAccessFile

empty file

write

move pointer
read

close file

```
18      // Retrieve the first number
19      inout.seek(0); // Move the file pointer to the beginning
20      System.out.println("The first number is " + inout.readInt());
21
22      // Retrieve the second number
23      inout.seek(1 * 4); // Move the file pointer to the second number
24      System.out.println("The second number is " + inout.readInt());
25
26      // Retrieve the tenth number
27      inout.seek(9 * 4); // Move the file pointer to the tenth number
28      System.out.println("The tenth number is " + inout.readInt());
29
30      // Modify the eleventh number
31      inout.writeInt(555);
32
33      // Append a new number
34      inout.seek(inout.length()); // Move the file pointer to the end
35      inout.writeInt(999);
36
37      // Display the new length
38      System.out.println("The new length is " + inout.length());
39
40      // Retrieve the new eleventh number
41      inout.seek(10 * 4); // Move the file pointer to the next number
42      System.out.println("The eleventh number is " + inout.readInt());
43
44      inout.close();
45    }
46  }
```



```
Current file length is 800
The first number is 0
The second number is 1
The tenth number is 9
The new length is 804
The eleventh number is 555
```

A `RandomAccessFile` is created for the file named **inout.dat** with mode "`rw`" to allow both read and write operations in line 6.

`inout.setLength(0)` sets the length to `0` in line 9. This, in effect, destroys the old contents of the file.

The `for` loop writes `200` `int` values from `0` to `199` into the file in lines 12–13. Since each `int` value takes `4` bytes, the total length of the file returned from `inout.length()` is now `800` (line 16), as shown in sample output.

Invoking `inout.seek(0)` in line 19 sets the file pointer to the beginning of the file. `inout.readInt()` reads the first value in line 20 and moves the file pointer to the next number. The second number is read in line 23.

`inout.seek(9 * 4)` (line 27) moves the file pointer to the tenth number. `inout.readInt()` reads the tenth number and moves the file pointer to the eleventh number in line 28. `inout.write(555)` writes a new eleventh number at the current position (line 31). The previous eleventh number is destroyed.

`inout.seek(inout.length())` moves the file pointer to the end of the file (line 34). `inout.writeInt(999)` writes a `999` to the file. Now the length of the file is increased by `4`, so `inout.length()` returns `804` (line 38).

`inout.seek(10 * 4)` moves the file pointer to the eleventh number in line 41. The new eleventh number, `555`, is displayed in line 42.

Key Terms

binary I/O 650
deserialization 664
file pointer 667
random-access file 667
sequential-access file 666
serialization 664
stream 650
text I/O 650

Chapter Summary

1. I/O can be classified into text I/O and binary I/O. Text I/O interprets data in sequences of characters. Binary I/O interprets data as raw binary values. How text is stored in a file depends on the encoding scheme for the file. Java automatically performs encoding and decoding for text I/O.

2. The `InputStream` and `OutputStream` classes are the roots of all binary I/O classes. `FileInputStream/FileOutputStream` associates a file for binary input/output. `BufferedInputStream`/`BufferedOutputStream` can be used to wrap on any binary I/O stream to improve performance. `DataInputStream`/`DataOutputStream` can be used to read/write primitive values and strings.

3. `ObjectInputStream`/`ObjectOutputStream` can be used to read/write objects in addition to primitive values and strings. To enable object serialization, the object's defining class must implement the `java.io.Serializable` marker interface.

4. The `RandomAccessFile` class enables you to read and write data to a file. You can open a file with the "`r`" mode to indicate that it is read-only, or with the "`rw`" mode to indicate that it is updateable. Since the `RandomAccessFile` class implements `DataInput` and `DataOutput` interfaces, many methods in `RandomAccessFile` are the same as those in `DataInputStream` and `DataOutputStream`.

Review Questions

Sections 19.1–19.2

19.1 What is a text file, and what is a binary file? Can you view a text file or a binary file using a text editor?

19.2 How do you read or write data in Java? What is a stream?

Section 19.3

19.3 What are the differences between text I/O and binary I/O?

19.4 How is a Java character represented in the memory, and how is a character represented in a text file?

19.5 If you write string "`ABC`" to an ASCII text file, what values are stored in the file?

19.6 If you write string "`100`" to an ASCII text file, what values are stored in the file? If you write a numeric byte-type value `100` using binary I/O, what values are stored in the file?

19.7 What is the encoding scheme for representing a character in a Java program? By default, what is the encoding scheme for a text file on Windows?

Section 19.4

19.8 Why do you have to declare to throw `IOException` in the method or use a try-catch block to handle `IOException` for Java IO programs?

19.9 Why should you always close streams?

19.10 `InputStream` reads bytes. Why does the `read()` method return an `int` instead of a `byte`? Find the abstract methods in `InputStream` and `OutputStream`.

19.11 Does `FileInputStream`/`FileOutputStream` introduce any new methods? How do you create a `FileInputStream`/`FileOutputStream`?

19.12 What will happen if you attempt to create an input stream on a nonexistent file? What will happen if you attempt to create an output stream on an existing file? Can you append data to an existing file?

19.13 How do you append data to an existing text file using `java.io.PrintWriter`?

19.14 Suppose a file contains an unspecified number of `double` values. Theses values were written to the file using the `writeDouble` method using a `DataOutputStream`. How do you write a program to read all these values? How do you detect the end of file?

19.15 What is written to a file using `writeByte(91)` on a `FileOutputStream`?

19.16 How do you check the end of a file in a binary input stream (`FileInputStream`, `DataInputStream`)?

19.17 What is wrong in the following code?

```
import java.io.*;

public class Test {
  public static void main(String[] args) {
    try {
      FileInputStream fis = new FileInputStream("test.dat");
    }
    catch (IOException ex) {
      ex.printStackTrace();
    }
    catch (FileNotFoundException ex) {
      ex.printStackTrace();
    }
  }
}
```

19.18 Suppose you run the program on Windows using the default ASCII encoding. After the program is finished, how many bytes are in the file **t.txt**? Show the contents of each byte.

```
public class Test {
  public static void main(String[] args)
      throws java.io.IOException {
    java.io.PrintWriter output =
      new java.io.PrintWriter("t.txt");
    output.printf("%s", "1234");
    output.printf("%s", "5678");
    output.close();
  }
}
```

19.19 After the program is finished, how many bytes are in the file **t.dat**? Show the contents of each byte.

```
import java.io.*;

public class Test {
  public static void main(String[] args) throws IOException {
    DataOutputStream output = new DataOutputStream(
```

```
      new FileOutputStream("t.dat"));
    output.writeInt(1234);
    output.writeInt(5678);
    output.close();
  }
}
```

19.20 For each of the following statements on a `DataOutputStream out`, how many bytes are sent to the output?

```
output.writeChar('A');
output.writeChars("BC");
output.writeUTF("DEF");
```

19.21 What are the advantages of using buffered streams? Are the following statements correct?

```
BufferedInputStream input1 =
  new BufferedInputStream(new FileInputStream("t.dat"));

DataInputStream input2 = new DataInputStream(
  new BufferedInputStream(new FileInputStream("t.dat")));

ObjectInputStream input3 = new ObjectInputStream(
  new BufferedInputStream(new FileInputStream("t.dat")));
```

Section 19.6

19.22 What types of objects can be stored using the `ObjectOutputStream`? What is the method for writing an object? What is the method for reading an object? What is the return type of the method that reads an object from `ObjectInputStream`?

19.23 If you serialize two objects of the same type, will they take the same amount of space? If not, give an example.

19.24 Is it true that any instance of `java.io.Serializable` can be successfully serialized? Are the static variables in an object serialized? How do you mark an instance variable not to be serialized?

19.25 Can you write an array to an `ObjectOutputStream`?

19.26 Is it true that `DataInputStream`/`DataOutputStream` can always be replaced by `ObjectInputStream`/`ObjectOutputStream`?

19.27 What will happen when you attempt to run the following code?

```
import java.io.*;

public class Test {
  public static void main(String[] args) throws IOException {
    ObjectOutputStream output =
      new ObjectOutputStream(new FileOutputStream("object.dat"));

    output.writeObject(new A());
  }
}

class A implements Serializable {
  B b = new B();
}

class B {
}
```

Section 19.7

19.28 Can `RandomAccessFile` streams read and write a data file created by `DataOutputStream`? Can `RandomAccessFile` streams read and write objects?

19.29 Create a `RandomAccessFile` stream for the file **address.dat** to allow the updating of student information in the file. Create a `DataOutputStream` for the file **address.dat**. Explain the differences between these two statements.

19.30 What happens if the file **test.dat** does not exist when you attempt to compile and run the following code?

```
import java.io.*;

public class Test {
  public static void main(String[] args) {
    try {
      RandomAccessFile raf =
        new RandomAccessFile("test.dat", "r");
      int i = raf.readInt();
    }
    catch (IOException ex) {
      System.out.println("IO exception");
    }
  }
}
```

Programming Exercises

Section 19.3

19.1* (*Creating a text file*) Write a program to create a file named **Exercise19_1.txt** if it does not exist. Append new data to it. Write 100 integers created randomly into the file using text I/O. Integers are separated by a space.

Section 19.4

19.2* (*Creating a binary data file*) Write a program to create a file named **Exercise19_2.dat** if it does not exist. Append new data to it. Write 100 integers created randomly into the file using binary I/O.

19.3* (*Summing all the integers in a binary data file*) Suppose a binary data file named **Exercise19_3.dat** has been created using `writeInt(int)` in `DataOutputStream`. The file contains an unspecified number of integers. Write a program to find the sum of integers.

19.4* (*Converting a text file into UTF*) Write a program that reads lines of characters from a text and writes each line as a UTF-8 string into a binary file. Display the sizes of the text file and the binary file. Use the following command to run the program:

```
java Exercise19_4 Welcome.java Welcome.utf
```

Section 19.6

19.5* (*Storing objects and arrays into a file*) Write a program that stores an array of five `int` values `1`, `2`, `3`, `4` and `5`, a `Date` object for current time, and a `double` value `5.5` into the file named **Exercise19_5.dat**.

19.6* (*Storing `Loan` objects*) The `Loan` class, in Listing 10.2, does not implement `Serializable`. Rewrite the `Loan` class to implement `Serializable`. Write a program that creates five `Loan` objects and stores them in a file named **Exercise19_6.dat**.

19.7* (*Restoring objects from a file*) Suppose a file named **Exercise19_7.dat** has been created using the `ObjectOutputStream`. The file contains `Loan` objects. The `Loan` class, in Listing 10.2, does not implement `Serializable`. Rewrite the `Loan` class to implement `Serializable`. Write a program that reads the `Loan` objects from the file and computes the total loan amount. Suppose you don't know how many `Loan` objects are in the file. Use `EOFException` to end the loop.

Section 19.7

19.8* (*Updating count*) Suppose you want to track how many times a program has been executed. You may store an `int` to count the file. Increase the count by `1` each time this program is executed. Let the program be **Exercise19_8** and store the count in **Exercise19_8.dat**.

19.9** (*Address book*) Supplement VII.B gives a case study of using random-access files for creating and manipulating an address book. Modify the case study by adding an *Update* button, as shown in Figure 19.18, to enable the user to modify the address that is being displayed.

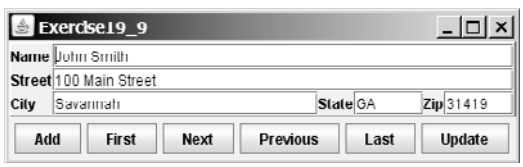


FIGURE 19.18 The application can store, retrieve, and update addresses from a file.

Comprehensive

19.10* (*Splitting files*) Suppose you wish to back up a huge file (e.g., a 10-GB AVI file) to a CD-R. You can achieve it by splitting the file into smaller pieces and backing up these pieces separately. Write a utility program that splits a large file into smaller ones using the following command:

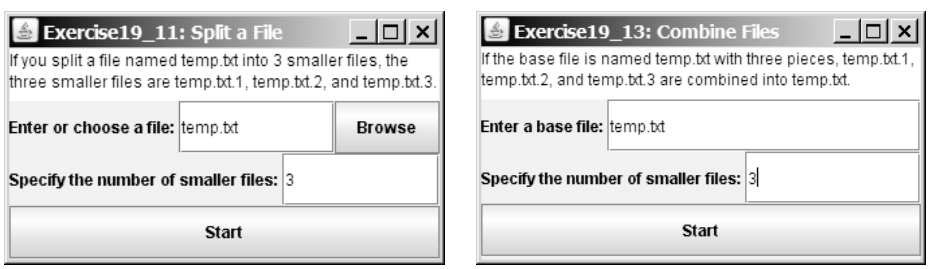


Video Note
Split a large file

```
java Exercise19_10 SourceFile numberOfPieces
```

The command creates files SourceFile.1, SourceFile.2, . . . , SourceFile.n, where `n` is `numberOfPieces` and the output files are about the same size.

19.11** (*Splitting files GUI*) Rewrite Exercise 19.10 with a GUI, as shown in Figure 19.19(a).


Exercise19_11: Split a File
If you split a file named temp.txt into 3 smaller files, the three smaller files are temp.txt.1, temp.txt.2, and temp.txt.3.
Enter or choose a file: temp.txt
Browse
Specify the number of smaller files: 3
Start
Exercise19_13: Combine Files
If the base file is named temp.txt with three pieces, temp.txt.1, temp.txt.2, and temp.txt.3 are combined into temp.txt.
Enter a base file: temp.txt
Specify the number of smaller files: 3
Start

(a) (b)

FIGURE 19.19 (a) The program splits a file. (b) The program combines files into a new file.

19.12* (*Combining files*) Write a utility program that combines the files together into a new file using the following command:

java Exercise19_12 SourceFile1 . . . SoureFilen TargetFile

The command combines SourceFile1, . . . , and SourceFilen into TargetFile.

19.13* (*Combining files GUI*) Rewrite Exercise 19.12 with a GUI, as shown in Figure 19.19(b).

19.14 (*Encrypting files*) Encode the file by adding `5` to every byte in the file. Write a program that prompts the user to enter an input file name and an output file name and saves the encrypted version of the input file to the output file.

19.15 (*Decrypting files*) Suppose a file is encrypted using the scheme in Exercise 19.14. Write a program to decode an encrypted file. Your program should prompt the user to enter an input file name and an output file name and should save the unencrypted version of the input file to the output file.

19.16 (*Frequency of characters*) Write a program that prompts the user to enter the name of an ASCII text file and display the frequency of the characters in the file.

19.17** (*`BitOutputStream`*) Implement a class named `BitOutputStream`, as shown in Figure 19.20, for writing bits to an output stream. The `writeBit(char bit)` method stores the bit in a byte variable. When you create a `BitOutputStream`, the byte is empty. After invoking `writeBit('1')`, the byte becomes `00000001`. After invoking `writeBit("0101")`, the byte becomes `00010101`. The first three bits are not filled yet. When a byte is full, it is sent to the output stream. Now the byte is reset to empty. You must close the stream by invoking the `close()` method. If the byte is not empty and not full, the `close()` method first fills the zeros to make a full `8` bits in the byte, and then output the byte and close the stream. For a hint, see Exercise 4.46. Write a test program that sends the bits `0100001001000001001101` to the file named **Exercise19_17.dat**.

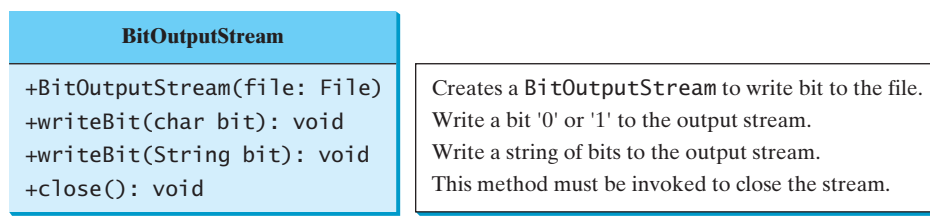

BitOutputStream
+BitOutputStream(file: File)
+writeBit(char bit): void
+writeBit(String bit): void
+close(): void
Creates a BitOutputStream to write bit to the file.
Write a bit '0' or '1' to the output stream.
Write a string of bits to the output stream.
This method must be invoked to close the stream.

FIGURE 19.20 `BitOutputStream` outputs a stream of bits to a file.

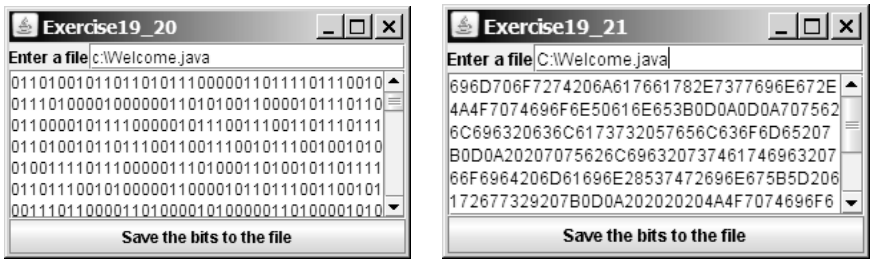
19.18* (*View bits*) Write the following method that displays the bit representation for the last byte in an integer:

```
public static String getBits(int value)
```

For a hint, see Exercise 4.46. Write a program that prompts the user to enter a file name, reads bytes from a file, and displays each byte's binary representation.

19.19* (*View hex*) Write a program that prompts the user to enter a file name, reads bytes from a file, and displays each byte's hex representation. (*Hint*: You may first convert the byte value into an 8-bit string, then convert the bit string into a two-digit hex string.)

19.20** (*Binary Editor*) Write a GUI application that lets the user enter a file name in the text field and press the *Enter* key to display its binary representation in a text area. The user can also modify the binary code and save it back to the file, as shown in Figure 19.21(a).



(a) (b)

FIGURE 19.21 The exercises enable the user to manipulate the contents of the file in binary and hex.

19.21** (*Hex Editor*) Write a GUI application that lets the user enter a file name in the text field and press the *Enter* key to display its hex representation in a text area. The user can also modify the hex code and save it back to the file, as shown in Figure 19.21(b).

This page intentionally left blank

CHAPTER 20

RECURSION

Objectives

- To describe what a recursive method is and the benefits of using recursion (§20.1).
- To develop recursive methods for recursive mathematical functions (§§20.2–20.3).
- To explain how recursive method calls are handled in a call stack (§§20.2–20.3).
- To use an overloaded helper method to derive a recursive method (§20.5).
- To solve selection sort using recursion (§20.5.1).
- To solve binary search using recursion (§20.5.2).
- To get the directory size using recursion (§20.6).
- To solve the Towers of Hanoi problem using recursion (§20.7).
- To draw fractals using recursion (§20.8).
- To solve the Eight Queens problem using recursion (§20.9).
- To discover the relationship and difference between recursion and iteration (§20.10).
- To know tail-recursive methods and why they are desirable (§20.11).



20.1 Introduction

search word problem

Suppose you want to find all the files under a directory that contain a particular word. How do you solve this problem? There are several ways to do so. An intuitive and effective solution is to use recursion by searching the files in the subdirectories recursively.

Eight Queens problem

The classic Eight Queens puzzle is to place eight queens on a chessboard such that no two can attack each other (i.e., no two queens are on the same row, same column, or same diagonal), as shown in Figure 20.1. How do you write a program to solve this problem? A good approach is to use recursion.

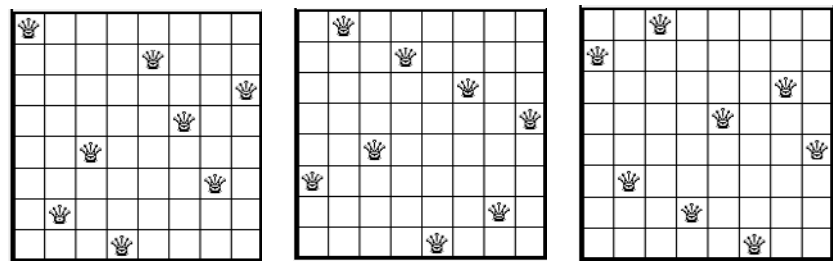


FIGURE 20.1 The Eight Queens problem can be solved using recursion.

recursive method

To use recursion is to program using *recursive methods*—methods that directly or indirectly invoke themselves. Recursion is a useful programming technique. In some cases, it enables you to develop a natural, straightforward, simple solution to an otherwise difficult problem. This chapter introduces the concepts and techniques of recursive programming and illustrates by examples how to "think recursively."

20.2 Problem: Computing Factorials

Many mathematical functions are defined using recursion. We begin with a simple example. The factorial of a number `n` can be recursively defined as follows:

```
0! = 1;
n! = n × (n - 1) × ... × 2 × 1 = n × (n - 1)!; n > 0
```

How do you find `n!` for a given `n`? To find `1!` is easy, because you know that `0!` is `1`, and `1!` is `1 × 0!`. Assuming that you know `(n - 1)!`, you can obtain `n!` immediately using `n × (n - 1)!`. Thus, the problem of computing `n!` is reduced to computing `(n - 1)!`. When computing `(n - 1)!`, you can apply the same idea recursively until `n` is reduced to `0`.

Let `factorial(n)` be the method for computing `n!`. If you call the method with `n = 0`, it immediately returns the result. The method knows how to solve the simplest case, which is referred to as the *base case* or the *stopping condition*. If you call the method with `n > 0`, it reduces the problem into a subproblem for computing the factorial of `n - 1`. The subproblem is essentially the same as the original problem, but is simpler or smaller. Because the subproblem has the same property as the original, you can call the method with a different argument, which is referred to as a *recursive call*.

base case or stopping condition

recursive call

The recursive algorithm for computing `factorial(n)` can be simply described as follows:

```
if (n == 0)
  return 1;
else
  return n * factorial(n - 1);
```

A recursive call can result in many more recursive calls, because the method keeps on dividing a subproblem into new subproblems. For a recursive method to terminate, the problem

must eventually be reduced to a stopping case, at which point the method returns a result to its caller. The caller then performs a computation and returns the result to its own caller. This process continues until the result is passed back to the original caller. The original problem can now be solved by multiplying `n` by the result of `factorial(n - 1)`.

Listing 20.1 gives a complete program that prompts the user to enter a nonnegative integer and displays the factorial for the number.

LISTING 20.1 ComputeFactorial.java

```
1  import java.util.Scanner;
2
3  public class ComputeFactorial {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8      System.out.print("Enter a nonnegative integer: ");
9      int n = input.nextInt();
10
11     // Display factorial
12     System.out.println("Factorial of " + n + " is " + factorial(n));
13   }
14
15   /** Return the factorial for a specified number */
16   public static long factorial(int n) {
17     if (n == 0) // Base case                                    base case
18       return 1;
19     else
20       return n * factorial(n - 1); // Recursive call           recursion
21   }
22 }
```

```
Enter a nonnegative integer: 4 Enter
Factorial of 4 is 24
```



```
Enter a nonnegative integer: 10 Enter
Factorial of 10 is 3628800
```



The `factorial` method (lines 16–21) is essentially a direct translation of the recursive mathematical definition for the factorial into Java code. The call to `factorial` is recursive because it calls itself. The parameter passed to `factorial` is decremented until it reaches the base case of `0`.

Figure 20.2 illustrates the execution of the recursive calls, starting with `n = 4`. The use of stack space for recursive calls is shown in Figure 20.3.



Caution

If recursion does not reduce the problem in a manner that allows it to eventually converge into the base case, infinite recursion can occur. For example, suppose you mistakenly write the `factorial` method as follows: infinite recursion

```
public static long factorial(int n) {
  return n * factorial(n - 1);
}
```

The method runs infinitely and causes a `StackOverflowError`.

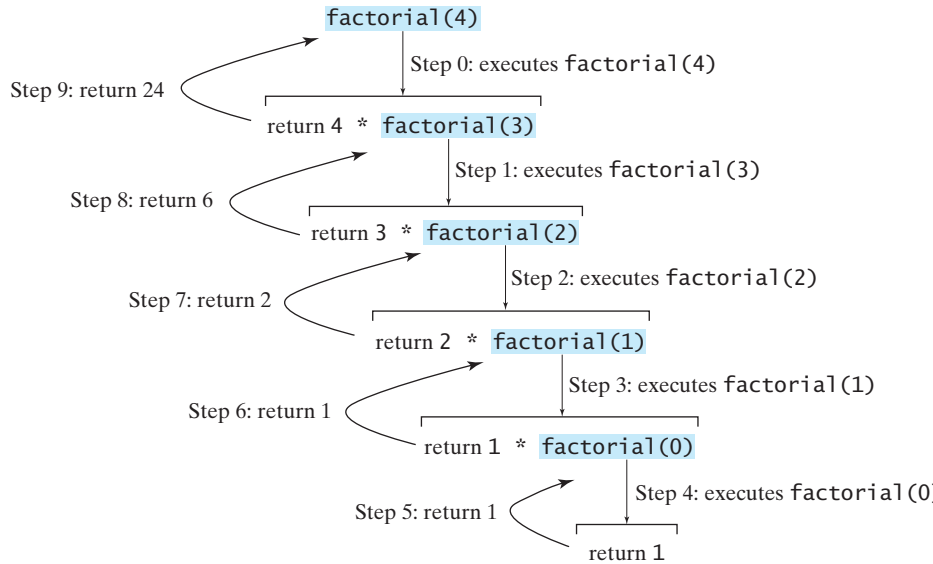


FIGURE 20.2 Invoking `factorial(4)` spawns recursive calls to `factorial`.

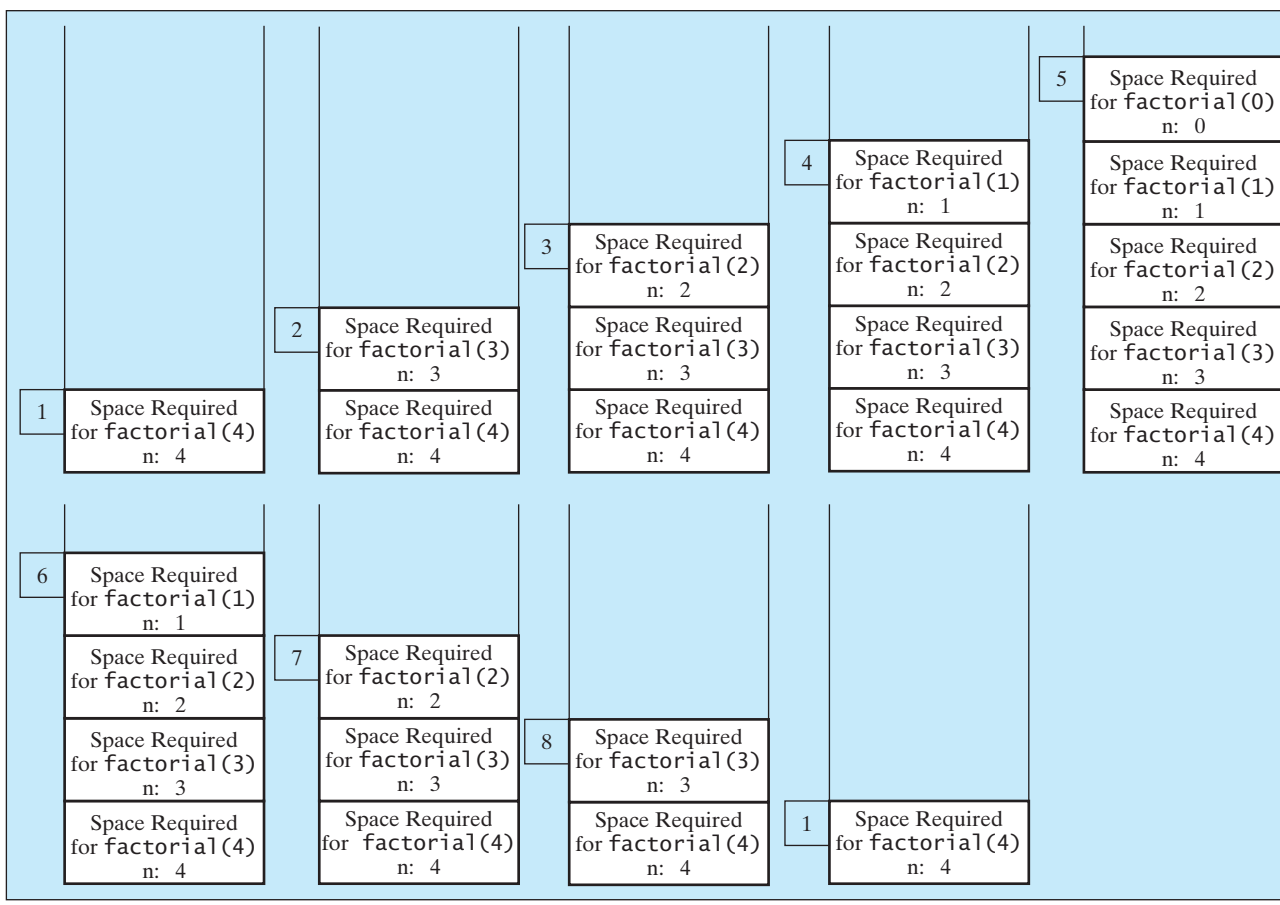


FIGURE 20.3 When `factorial(4)` is being executed, the `factorial` method is called recursively, causing stack space to dynamically change.



Pedagogical Note

It is simpler and more efficient to implement the `factorial` method using a loop. However, we use the recursive `factorial` method here to demonstrate the concept of recursion. Later in this chapter, we will present some problems that are inherently recursive and are difficult to solve without using recursion.

20.3 Problem: Computing Fibonacci Numbers

The `factorial` method in the preceding section could easily be rewritten without using recursion. In some cases, however, using recursion enables you to give a natural, straightforward, simple solution to a program that would otherwise be difficult to solve. Consider the well-known Fibonacci-series problem:

The series:	0	1	1	2	3	5	8	13	21	34	55	89	...
indices:	0	1	2	3	4	5	6	7	8	9	10	11	

The Fibonacci series begins with `0` and `1`, and each subsequent number is the sum of the preceding two. The series can be recursively defined as follows:

```
fib(0) = 0;
fib(1) = 1;
fib(index) = fib(index - 2) + fib(index - 1); index >= 2
```

The Fibonacci series was named for Leonardo Fibonacci, a medieval mathematician, who originated it to model the growth of the rabbit population. It can be applied in numeric optimization and in various other areas.

How do you find `fib(index)` for a given `index`? It is easy to find `fib(2)`, because you know `fib(0)` and `fib(1)`. Assuming that you know `fib(index - 2)` and `fib(index - 1)`, you can obtain `fib(index)` immediately. Thus, the problem of computing `fib(index)` is reduced to computing `fib(index - 2)` and `fib(index - 1)`. When doing so, you apply the idea recursively until `index` is reduced to `0` or `1`.

The base case is `index = 0` or `index = 1`. If you call the method with `index = 0` or `index = 1`, it immediately returns the result. If you call the method with `index >= 2`, it divides the problem into two subproblems for computing `fib(index - 1)` and `fib(index - 2)` using recursive calls. The recursive algorithm for computing `fib(index)` can be simply described as follows:

```
if (index == 0)
  return 0;
else if (index == 1)
  return 1;
else
  return fib(index - 1) + fib(index - 2);
```

Listing 20.2 gives a complete program that prompts the user to enter an index and computes the Fibonacci number for the index.

LISTING 20.2 ComputeFibonacci.java

```
1 import java.util.Scanner;
2
3 public class ComputeFibonacci {
4   /** Main method */
5   public static void main(String args[]) {
6     // Create a Scanner
7     Scanner input = new Scanner(System.in);
8     System.out.print("Enter an index for the Fibonacci number: ");
```

```
 9     int index = input.nextInt();
10
11     // Find and display the Fibonacci number
12     System.out.println(
13       "Fibonacci number at index " + index + " is " + fib(index));
14   }
15
16   /** The method for finding the Fibonacci number */
17   public static long fib(long index) {
18     if (index == 0) // Base case
19       return 0;
20     else if (index == 1) // Base case
21       return 1;
22     else  // Reduction and recursive calls
23       return fib(index - 1) + fib(index - 2);
24   }
25 }
```

base case

base case

recursion



```
Enter an index for the Fibonacci number: 1
Fibonacci number at index 1 is 1
```



```
Enter an index for the Fibonacci number: 6 Enter
Fibonacci number at index 6 is 8
```



```
Enter an index for the Fibonacci number: 7 Enter
Fibonacci number at index 7 is 13
```

The program does not show the considerable amount of work done behind the scenes by the computer. Figure 20.4, however, shows successive recursive calls for evaluating `fib(4)`. The original method, `fib(4)`, makes two recursive calls, `fib(3)` and `fib(2)`, and then returns `fib(3) + fib(2)`. But in what order are these methods called? In Java, operands are evaluated from left to right. `fib(2)` is called after `fib(3)` is completely evaluated. The labels in Figure 20.4 show the order in which methods are called.

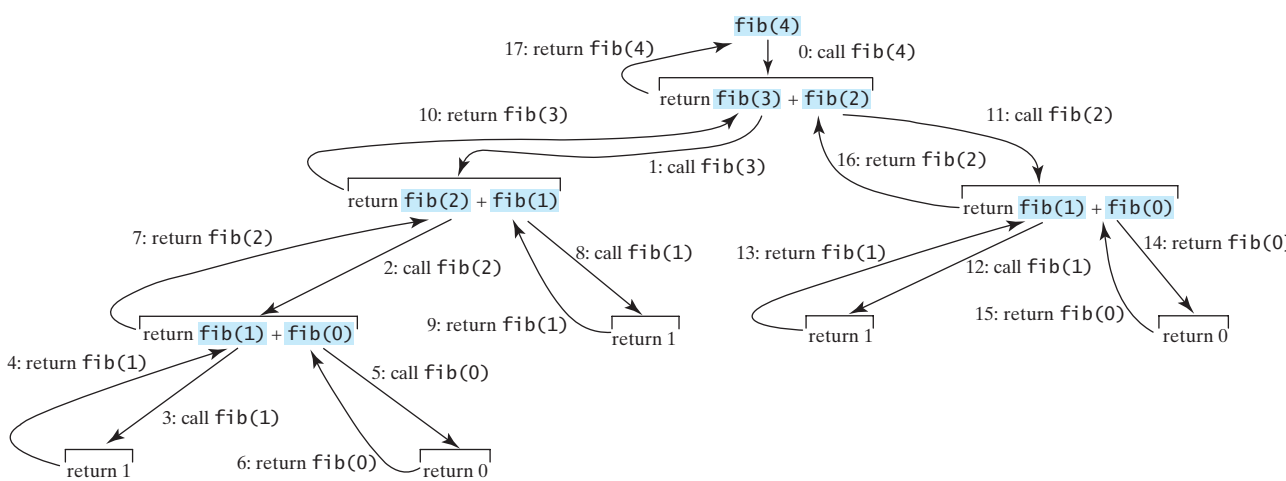


FIGURE 20.4 Invoking `fib(4)` spawns recursive calls to `fib`.

As shown in Figure 20.4, there are many duplicated recursive calls. For instance, `fib(2)` is called twice, `fib(1)` three times, and `fib(0)` twice. In general, computing `fib(index)` requires roughly twice as many recursive calls as does computing `fib(index - 1)`. As you try larger index values, the number of calls substantially increases.

Besides the large number of recursive calls, the computer requires more time and space to run recursive methods.



Pedagogical Note

The recursive implementation of the `fib` method is very simple and straightforward, but not efficient. See Exercise 20.2 for an efficient solution using loops. Though it is not practical, the recursive `fib` method is a good example of how to write recursive methods.

20.4 Problem Solving Using Recursion

recursion characteristics

The preceding sections presented two classic recursion examples. All recursive methods have the following characteristics:

- The method is implemented using an `if-else` or a `switch` statement that leads to different cases. — if-else
- One or more base cases (the simplest case) are used to stop recursion. — base cases
- Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case. — reduction

In general, to solve a problem using recursion, you break it into subproblems. Each subproblem is almost the same as the original problem but smaller in size. You can apply the same approach to each subproblem to solve it recursively.

Let us consider the simple problem of printing a message `n` times. You can break the problem into two subproblems: one is to print the message one time and the other is to print it `n - 1` times. The second problem is the same as the original problem but smaller in size. The base case for the problem is `n == 0`. You can solve this problem using recursion as follows:

```
public static void nPrintln(String message, int times) {
  if (times >= 1) {
    System.out.println(message);
    nPrintln(message, times - 1);
  } // The base case is times == 0
}
```

recursive call

Note that the `fib` method in the preceding example returns a value to its caller, but the `nPrintln` method is `void` and does not.

think recursively

If you *think recursively*, you can use recursion to solve many of the problems presented in earlier chapters of this book. Consider the palindrome problem in Listing 9.1. Recall that a string is a palindrome if it reads the same from the left and from the right. For example, mom and dad are palindromes, but uncle and aunt are not. The problem of checking whether a string is a palindrome can be divided into two subproblems:

- Check whether the first character and the last character of the string are equal.
- Ignore the two end characters and check whether the rest of the substring is a palindrome.

The second subproblem is the same as the original problem but smaller in size. There are two base cases: (1) the two end characters are not same; (2) the string size is `0` or `1`. In case 1, the string is not a palindrome; and in case 2, the string is a palindrome. The recursive method for this problem can be implemented as shown in Listing 20.3.

LISTING 20.3 RecursivePalindromeUsingSubstring.java

```
1  public class RecursivePalindromeUsingSubstring {
2    public static boolean isPalindrome(String s) {
3      if (s.length() <= 1) // Base case
4        return true;
5      else if (s.charAt(0) != s.charAt(s.length() - 1)) // Base case
6        return false;
7      else
8        return isPalindrome(s.substring(1, s.length() - 1));
9    }
10
11   public static void main(String[] args) {
12     System.out.println("Is moon a palindrome? "
13       + isPalindrome("moon"));
14     System.out.println("Is noon a palindrome? "
15       + isPalindrome("noon"));
16     System.out.println("Is a a palindrome? " + isPalindrome("a"));
17     System.out.println("Is aba a palindrome? " +
18       isPalindrome("aba"));
19     System.out.println("Is ab a palindrome? " + isPalindrome("ab"));
20   }
21 }
```

method header
base case
base case
recursive call



```
Is moon a palindrome? false
Is noon a palindrome? true
Is a a palindrome? true
Is aba a palindrome? true
Is ab a palindrome? false
```

The `substring` method in line 8 creates a new string that is the same as the original string except without the first and last characters. Checking whether a string is a palindrome is equivalent to checking whether the substring is a palindrome if the two end characters in the original string are the same.

20.5 Recursive Helper Methods

The preceding recursive `isPalindrome` method is not efficient, because it creates a new string for every recursive call. To avoid creating new strings, you can use the low and high indices to indicate the range of the substring. These two indices must be passed to the recursive method. Since the original method is `isPalindrome(String s)`, you have to create a new method `isPalindrome(String s, int low, int high)` to accept additional information on the string, as shown in Listing 20.4.

LISTING 20.4 RecursivePalindrome.java

```
1  public class RecursivePalindrome {
2    public static boolean isPalindrome(String s) {
3      return isPalindrome(s, 0, s.length() - 1);
4    }
5
6    public static boolean isPalindrome(String s, int low, int high) {
7      if (high <= low) // Base case
8        return true;
9      else if (s.charAt(low) != s.charAt(high)) // Base case
10       return false;
```

helper method
base case
base case

```
11      else
12        return isPalindrome(s, low + 1, high - 1);
13    }
14
15    public static void main(String[] args) {
16      System.out.println("Is moon a palindrome? "
17        + isPalindrome("moon"));
18      System.out.println("Is noon a palindrome? "
19        + isPalindrome("noon"));
20      System.out.println("Is a a palindrome? " + isPalindrome("a"));
21      System.out.println("Is aba a palindrome? " + isPalindrome("aba"));
22      System.out.println("Is ab a palindrome? " + isPalindrome("ab"));
23    }
24 }
```

Two overloaded `isPalindrome` methods are defined. The first, `isPalindrome(String s)`, checks whether a string is a palindrome, and the second, `isPalindrome(String s, int low, int high)`, checks whether a substring `s(low..high)` is a palindrome. The first method passes the string `s` with `low = 0` and `high = s.length() - 1` to the second method. The second method can be invoked recursively to check a palindrome in an ever-shrinking substring. It is a common design technique in recursive programming to define a second method that receives additional parameters. Such a method is known as a *recursive helper method*.

recursive helper method

Helper methods are very useful in designing recursive solutions for problems involving strings and arrays. The sections that follow give two more examples.

20.5.1 Selection Sort

Selection sort was introduced in §6.10.1, "Selection Sort." Recall that it finds the smallest number in the list and places it first. It then finds the smallest number remaining and places it after the first, and so on until the remaining list contains only a single number. The problem can be divided into two subproblems:

- Find the smallest number in the list and swap it with the first number.
- Ignore the first number and sort the remaining smaller list recursively.

The base case is that the list contains only one number. Listing 20.5 gives the recursive sort method.

LISTING 20.5 RecursiveSelectionSort.java

```
1 public class RecursiveSelectionSort {
2   public static void sort(double[] list) {
3     sort(list, 0, list.length - 1); // Sort the entire list
4   }
5
6   public static void sort(double[] list, int low, int high) {
7     if (low < high) {
8       // Find the smallest number and its index in list(low .. high)
9       int indexOfMin = low;
10      double min = list[low];
11      for (int i = low + 1; i <= high; i++) {
12        if (list[i] < min) {
13          min = list[i];
14          indexOfMin = i;
15        }
16      }
17
18      // Swap the smallest in list(low .. high) with list(low)
```

helper method

base case

```
19        list[indexOfMin] = list[low];
20        list[low] = min;
21
22        // Sort the remaining list(low+1 .. high)
23        sort(list, low + 1, high);
24      }
25    }
26  }
```

recursive call

Two overloaded `sort` methods are defined. The first method, `sort(double[] list)`, sorts an array in `list[0..list.length - 1]` and the second method `sort(double[] list, int low, int high)` sorts an array in `list[low..high]`. The second method can be invoked recursively to sort an ever-shrinking subarray.

20.5.2 Binary Search



Video Note
Binary search

Binary search was introduced in §6.9.2. For binary search to work, the elements in the array must already be ordered. The binary search first compares the key with the element in the middle of the array. Consider the following three cases:

- Case 1: If the key is less than the middle element, recursively search the key in the first half of the array.
- Case 2: If the key is equal to the middle element, the search ends with a match.
- Case 3: If the key is greater than the middle element, recursively search the key in the second half of the array.

Case 1 and Case 3 reduce the search to a smaller list. Case 2 is a base case when there is a match. Another base case is that the search is exhausted without a match. Listing 20.6 gives a clear, simple solution for the binary search problem using recursion.

LISTING 20.6 Recursive Binary Search Method

```
1  public class RecursiveBinarySearch {
2    public static int recursiveBinarySearch(int[] list, int key) {
3      int low = 0;
4      int high = list.length - 1;
5      return recursiveBinarySearch(list, key, low, high);
6    }
7
8    public static int recursiveBinarySearch(int[] list, int key,
9        int low, int high) {
10     if (low > high)  // The list has been exhausted without a match
11       return -low - 1;
12
13     int mid = (low + high) / 2;
14     if (key < list[mid])
15       return recursiveBinarySearch(list, key, low, mid - 1);
16     else if (key == list[mid])
17       return mid;
18     else
19       return recursiveBinarySearch(list, key, mid + 1, high);
20   }
21 }
```

helper method

base case

recursive call

base case

recursive call

The first method finds a key in the whole list. The second method finds a key in the list with index from `low` to `high`.

The first `binarySearch` method passes the initial array with `low = 0` and `high = list.length - 1` to the second `binarySearch` method. The second method is invoked recursively to find the key in an ever-shrinking subarray.

20.6 Problem: Finding the Directory Size



Video Note
Directory size

The preceding examples can easily be solved without using recursion. This section presents a problem that is difficult to solve without using recursion. The problem is to find the size of a directory. The size of a directory is the sum of the sizes of all files in the directory. A directory d may contain subdirectories. Suppose a directory contains files $f_1, f_2, \ldots, f_m$ and subdirectories $d_1, d_2, \ldots, d_n$, as shown in Figure 20.5.

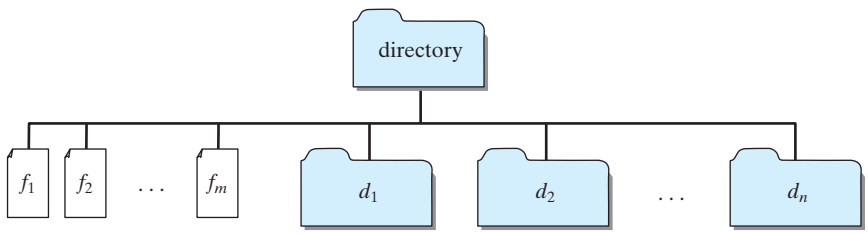


FIGURE 20.5 A directory contains files and subdirectories.

The size of the directory can be defined recursively as follows:

$$size(d) = size(f_1) + size(f_2) + \ldots + size(f_m) + size(d_1) + size(d_2) + \ldots + size(d_n)$$

The `File` class, introduced in §9.6, can be used to represent a file or a directory and obtain the properties for files and directories. Two methods in the `File` class are useful for this problem:

- The `length()` method returns the size of a file.
- The `listFiles()` method returns an array of `File` objects under a directory.

Listing 20.7 gives a program that prompts the user to enter a directory or a file and displays its size.

LISTING 20.7 DirectorySize.java

```
1  import java.io.File;
2  import java.util.Scanner;
3
4  public class DirectorySize {
5    public static void main(String[] args) {
6      // Prompt the user to enter a directory or a file
7      System.out.print("Enter a directory or a file: ");
8      Scanner input = new Scanner(System.in);
9      String directory = input.nextLine();
10
11     // Display the size
12     System.out.println(getSize(new File(directory)) + " bytes");      invoke method
13   }
14
15   public static long getSize(File file) {                            getSize method
16     long size = 0; // Store the total size of all files
17
18     if (file.isDirectory()) {                                        is directory?
```

all subitems

recursive call

base case

```
19        File[] files = file.listFiles(); // All files and subdirectories
20        for (int i = 0; i < files.length; i++) {
21          size += getSize(files[i]); // Recursive call
22        }
23      }
24      else { // Base case
25        size += file.length();
26      }
27
28      return size;
29    }
30  }
```



```
Enter a directory or a file: c:\book Enter
48619631 bytes
```



```
Enter a directory or a file: c:\book\Welcome.java Enter
172 bytes
```



```
Enter a directory or a file: c:\book\NonExistentFile Enter
0 bytes
```

If the `file` object represents a directory (line 18), each subitem (file or subdirectory) in the directory is recursively invoked to obtain its size (line 21). If the `file` object represents a file (line 24), the file size is obtained (line 25).

What happens if an incorrect or a nonexistent directory is entered? The program will detect that it is not a directory and invoke `file.length()` (line 25), which returns `0`. So, in this case, the `getSize` method will return `0`.

testing base cases



Tip

To avoid mistakes, it is a good practice to test base cases. For example, you should test the program for an input of file, an empty directory, a nonexistent directory, and a nonexistent file.

20.7 Problem: Towers of Hanoi

The Towers of Hanoi problem is a classic problem that can be solved easily using recursion but is difficult to solve otherwise.

The problem involves moving a specified number of disks of distinct sizes from one tower to another while observing the following rules:

- There are n disks labeled 1, 2, 3, . . . , n and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the top disk on a tower.

The objective of the problem is to move all the disks from A to B with the assistance of C. For example, if you have three disks, the steps to move all of the disks from A to B are shown in Figure 20.6.

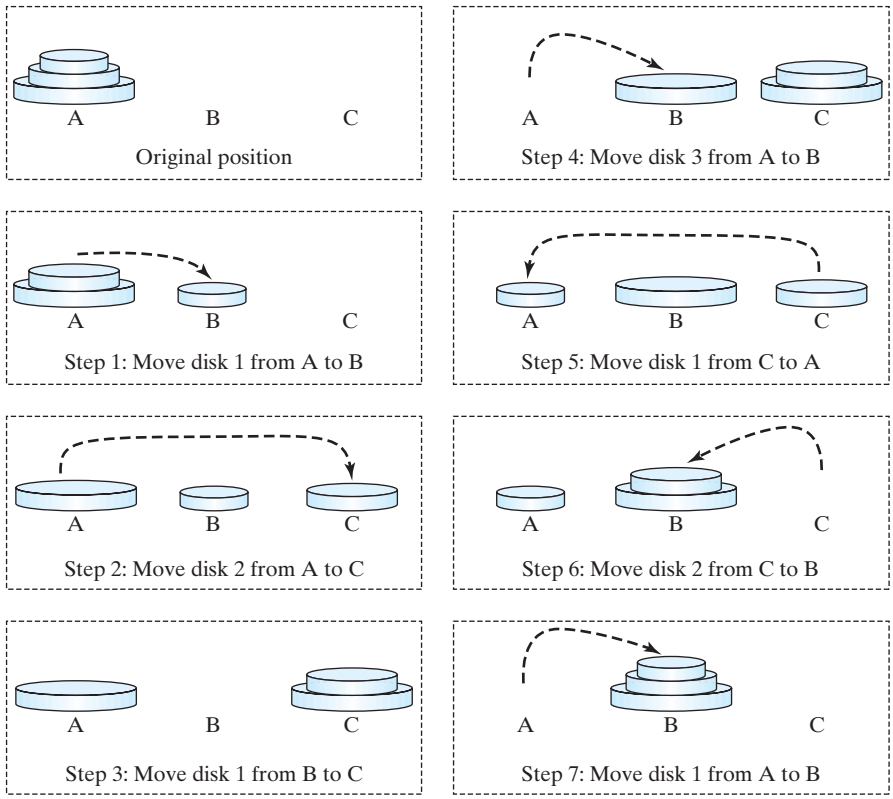


FIGURE 20.6 The goal of the Towers of Hanoi problem is to move disks from tower A to tower B without breaking the rules.



Note

The Towers of Hanoi is a classic computer-science problem, to which many Websites are devoted. One of them worth looking at is www.cut-the-knot.com/recurrence/hanoi.html.

In the case of three disks, you can find the solution manually. For a larger number of disks, however—even for four—the problem is quite complex. Fortunately, the problem has an inherently recursive nature, which leads to a straightforward recursive solution.

The base case for the problem is `n = 1`. If `n == 1`, you could simply move the disk from A to B. When `n > 1`, you could split the original problem into three subproblems and solve them sequentially.

1. Move the first `n - 1` disks from A to C with the assistance of tower B, as shown in Step 1 in Figure 20.7.
2. Move disk `n` from A to B, as shown in Step 2 in Figure 20.7.
3. Move `n - 1` disks from C to B with the assistance of tower A, as shown in Step 3 in Figure 20.7.

The following method moves *n* disks from the `fromTower` to the `toTower` with the assistance of the `auxTower`:

```
void moveDisks(int n, char fromTower, char toTower, char auxTower)
```

The algorithm for the method can be described as follows:

```
if (n == 1) // Stopping condition
  Move disk 1 from the fromTower to the toTower;
else {
```

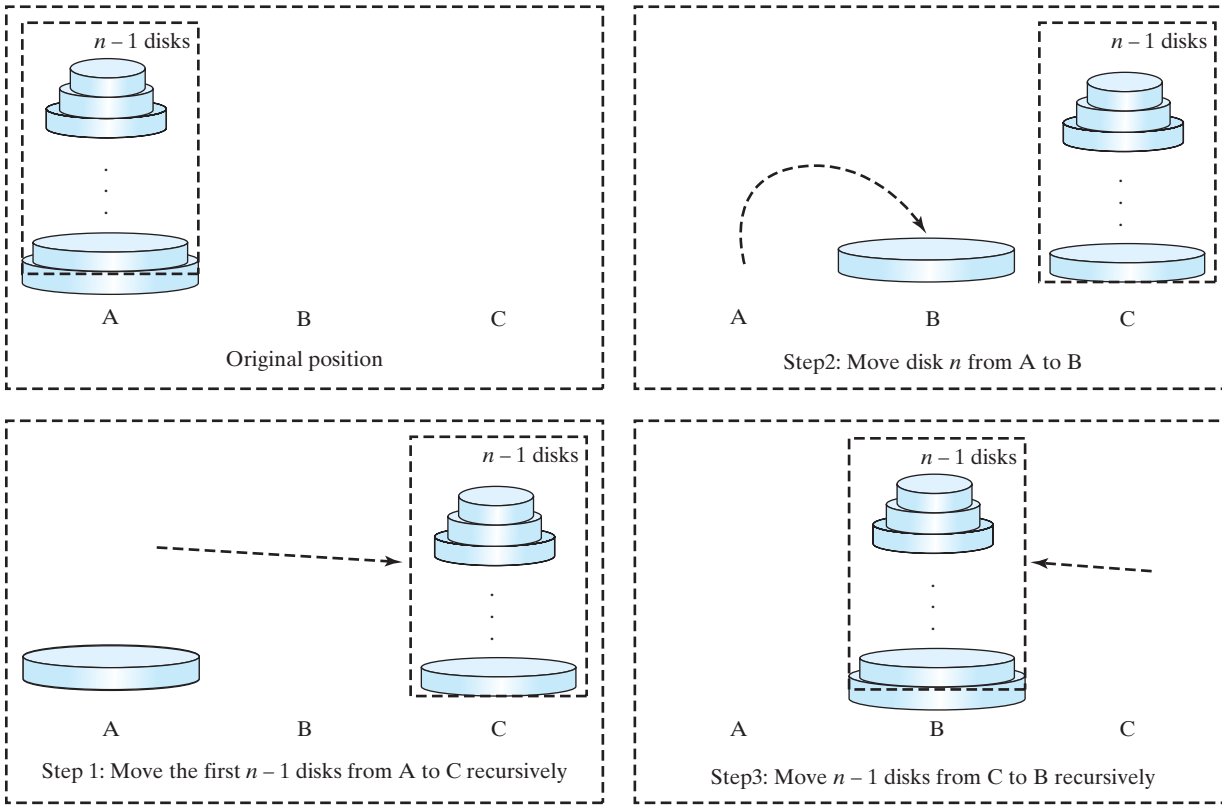


FIGURE 20.7 The Towers of Hanoi problem can be decomposed into three subproblems.

```
    moveDisks(n - 1, fromTower, auxTower, toTower);
    Move disk n from the fromTower to the toTower;
    moveDisks(n - 1, auxTower, toTower, fromTower);
  }
```

Listing 20.8 gives a program that prompts the user to enter the number of disks and invokes the recursive method `moveDisks` to display the solution for moving the disks.

LISTING 20.8 TowersOfHanoi.java

```
1  import java.util.Scanner;
2
3  public class TowersOfHanoi {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8      System.out.print("Enter number of disks: ");
9      int n = input.nextInt();
10
11     // Find the solution recursively
12     System.out.println("The moves are:");
13     moveDisks(n, 'A', 'B', 'C');
14   }
15
16   /** The method for finding the solution to move n disks
17       from fromTower to toTower with auxTower */
```

```
18   public static void moveDisks(int n, char fromTower,
19       char toTower, char auxTower) {
20     if (n == 1) // Stopping condition                          base case
21       System.out.println("Move disk " + n + " from " +
22         fromTower + " to " + toTower);
23     else {
24       moveDisks(n - 1, fromTower, auxTower, toTower);          recursion
25       System.out.println("Move disk " + n + " from " +
26         fromTower + " to " + toTower);
27       moveDisks(n - 1, auxTower, toTower, fromTower);          recursion
28     }
29   }
30 }
```

```
Enter number of disks: 4 Enter
The moves are:
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
Move disk 4 from A to B
Move disk 1 from C to B
Move disk 2 from C to A
Move disk 1 from B to A
Move disk 3 from C to B
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
```



This problem is inherently recursive. Using recursion makes it possible to find a natural, simple solution. It would be difficult to solve the problem without using recursion.

Consider tracing the program for `n = 3`. The successive recursive calls are shown in Figure 20.8. As you can see, writing the program is easier than tracing the recursive calls. The system uses stacks to trace the calls behind the scenes. To some extent, recursion provides a level of abstraction that hides iterations and other details from the user.

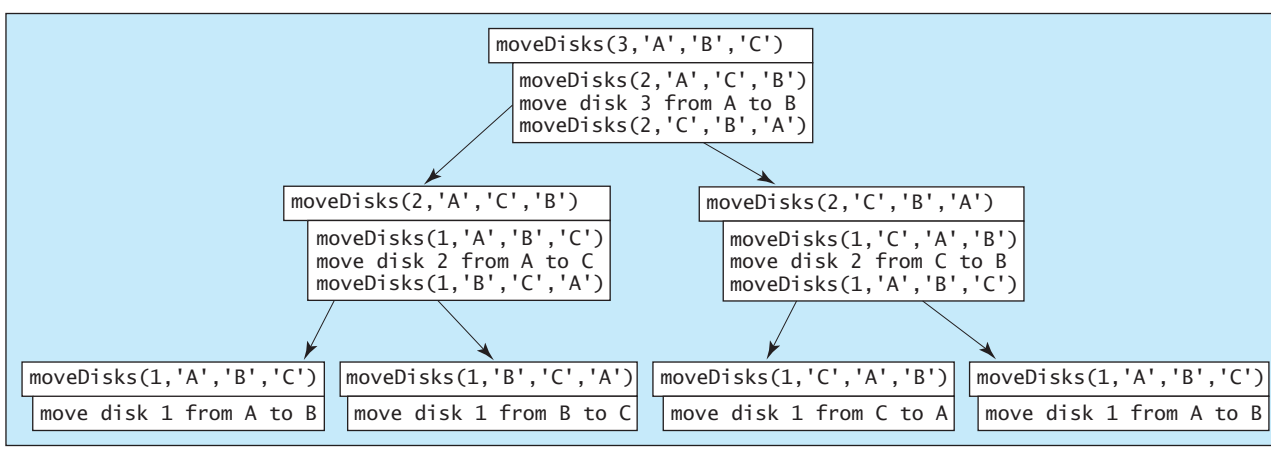


FIGURE 20.8 Invoking `moveDisks(3, 'A', 'B', 'C')` spawns calls to `moveDisks` recursively.

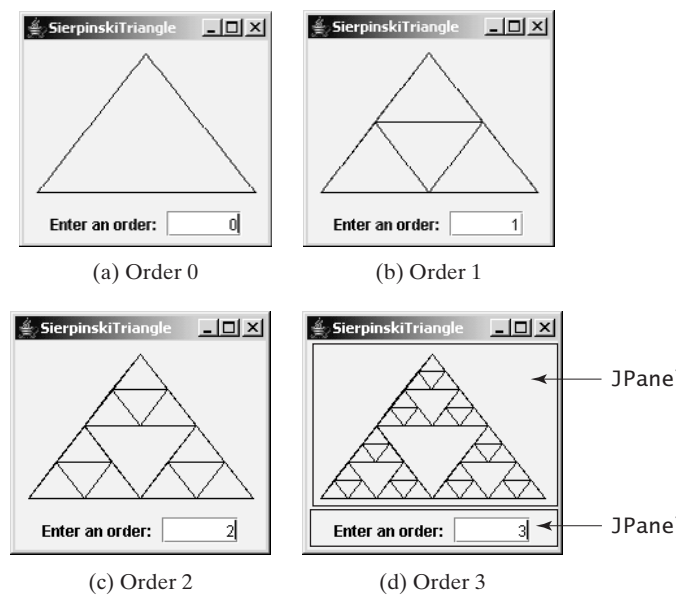
20.8 Problem: Fractals

Video Note
Fractal (Sierpinski triangle)

A *fractal* is a geometrical figure, but unlike triangles, circles, and rectangles, fractals can be divided into parts, each a reduced-size copy of the whole. There are many interesting examples of fractals. This section introduces a simple fractal, the *Sierpinski triangle*, named after a famous Polish mathematician.

A Sierpinski triangle is created as follows:

1. Begin with an equilateral triangle, which is considered to be a Sierpinski fractal of order (or level) **0**, as shown in Figure 20.9(a).
2. Connect the midpoints of the sides of the triangle of order **0** to create a Sierpinski triangle of order **1** (Figure 20.9(b)).
3. Leave the center triangle intact. Connect the midpoints of the sides of the three other triangles to create a Sierpinski triangle of order **2** (Figure 20.9(c)).
4. You can repeat the same process recursively to create a Sierpinski triangle of order **3**, **4**, . . . , and so on (Figure 20.9(d)).



(a) Order 0 (b) Order 1

(c) Order 2 (d) Order 3

FIGURE 20.9 A Sierpinski triangle is a pattern of recursive triangles.

The problem is inherently recursive. How do you develop a recursive solution for it? Consider the base case when the order is **0**. It is easy to draw a Sierpinski triangle of order **0**. How do you draw a Sierpinski triangle of order **1**? The problem can be reduced to drawing three Sierpinski triangles of order **0**. How do you draw a Sierpinski triangle of order **2**? The problem can be reduced to drawing three Sierpinski triangles of order **1**. So the problem of drawing a Sierpinski triangle of order ***n*** can be reduced to drawing three Sierpinski triangles of order ***n*** **- 1**.

Listing 20.9 gives a Java applet that displays a Sierpinski triangle of any order, as shown in Figure 20.9. You can enter an order in a text field to display a Sierpinski triangle of the specified order.

LISTING 20.9 SierpinskiTriangle.java

```
1 import javax.swing.*;
2 import java.awt.*;
3 import java.awt.event.*;
```

```
 4
 5 public class SierpinskiTriangle extends JApplet {
 6   private JTextField jtfOrder = new JTextField("0", 5); // Order
 7   private SierpinskiTrianglePanel trianglePanel =
 8     new SierpinskiTrianglePanel(); // To display the pattern
 9
10   public SierpinskiTriangle() {
11     // Panel to hold label, text field, and a button
12     JPanel panel = new JPanel();
13     panel.add(new JLabel("Enter an order: "));
14     panel.add(jtfOrder);
15     jtfOrder.setHorizontalAlignment(SwingConstants.RIGHT);
16
17     // Add a Sierpinski triangle panel to the applet
18     add(trianglePanel);
19     add(panel, BorderLayout.SOUTH);
20
21     // Register a listener
22     jtfOrder.addActionListener(new ActionListener() {                    listener
23       public void actionPerformed(ActionEvent e) {
24         trianglePanel.setOrder(Integer.parseInt(jtfOrder.getText()));    set a new order
25       }
26     });
27   }
28
29   static class SierpinskiTrianglePanel extends JPanel {
30     private int order = 0;
31
32     /** Set a new order */
33     public void setOrder(int order) {
34       this.order = order;
35       repaint();
36     }
37
38     protected void paintComponent(Graphics g) {
39       super.paintComponent(g);
40
41       // Select three points in proportion to the panel size
42       Point p1 = new Point(getWidth() / 2, 10);                          three initial points
43       Point p2 = new Point(10, getHeight() - 10);
44       Point p3 = new Point(getWidth() - 10, getHeight() - 10);
45
46       displayTriangles(g, order, p1, p2, p3);
47     }
48
49     private static void displayTriangles(Graphics g, int order,
50         Point p1, Point p2, Point p3) {
51       if (order >= 0) {
52         // Draw a triangle to connect three points
53         g.drawLine(p1.x, p1.y, p2.x, p2.y);                              draw a triangle
54         g.drawLine(p1.x, p1.y, p3.x, p3.y);
55         g.drawLine(p2.x, p2.y, p3.x, p3.y);
56
57         // Get the midpoint on each edge in the triangle
58         Point p12 = midpoint(p1, p2);
59         Point p23 = midpoint(p2, p3);
60         Point p31 = midpoint(p3, p1);
61
62         // Recursively display three triangles
63         displayTriangles(g, order - 1, p1, p12, p31);                    top subtriangle
```

left subtriangle
right subtriangle

```
64          displayTriangles(g, order - 1, p12, p2, p23);
65          displayTriangles(g, order - 1, p31, p23, p3);
66        }
67      }
68
69      private static Point midpoint(Point p1, Point p2) {
70        return new Point((p1.x + p2.x) / 2, (p1.y + p2.y) / 2);
71      }
72    }
```

main method omitted

```
73 }
```

The initial triangle has three points set in proportion to the panel size (lines 42–44). The `displayTriangles(g, order, p1, p2, p3)` method (lines 49–67) performs the following tasks if `order >= 0`:

1. Display a triangle to connect three points `p1`, `p2`, and `p3` in lines 53–55 , as shown in Figure 20.10(a).
2. Obtain a midpoint between `p1` and `p2` (line 58), a midpoint between `p2` and `p3` (line 59), and a midpoint between `p3` and `p1` (line 60), as shown in Figure 20.10(b).
3. Recursively invoke `displayTriangles` with a reduced order to display three smaller Sierpinski triangles (lines 63–66). Note each small Sierpinski triangle is structurally identical to the original big Sierpinski triangle except that the order of a small triangle is one less, as shown in Figure 20.10(b).

A Sierpinski triangle is displayed in a `SierpinskiTrianglePanel`. The `order` property in the inner class `SierpinskiTrianglePanel` specifies the order for the Sierpinski triangle. The `Point` class, introduced in §16.10, "Mouse Events," represents a point on a component. The `midpoint(Point p1, Point p2)` method returns the midpoint between `p1` and `p2` (lines 72–74).

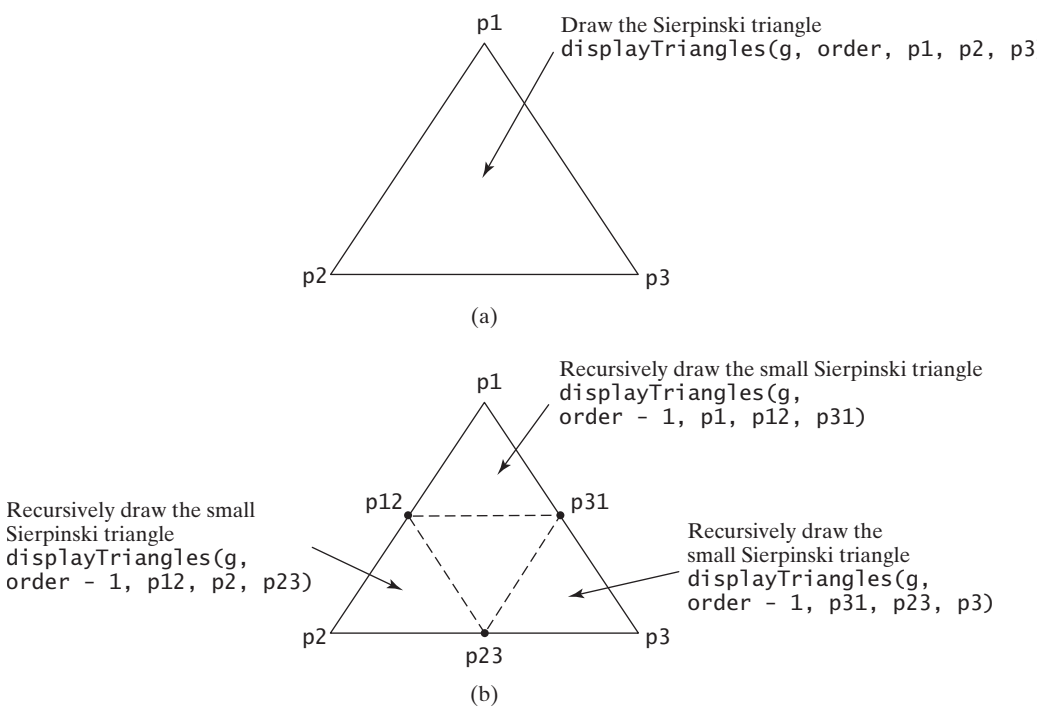


FIGURE 20.10 Drawing a Sierpinski triangle spawns calls to draw three small Sierpinski triangles recursively.

20.9 Problem: Eight Queens

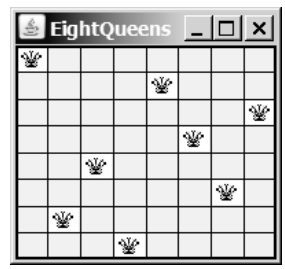
This section gives a recursive solution to the Eight Queens problem presented at the beginning of the chapter. The task is to find a solution to place a queen in each row on a chessboard such that no two queens can attack each other. You may use a two-dimensional array to represent a chessboard. However, since each row can have only one queen, it is sufficient to use a one-dimensional array to denote the position of the queen in the row. So, you may define array `queens` as follows:

```
int[] queens = new int[8];
```

Assign `j` to `queens[i]` to denote that a queen is placed in row `i` and column `j`. Figure 20.11(a) shows the contents of array `queens` for the chessboard in Figure 20.11(b).

queens[0]	0
queens[1]	4
queens[2]	7
queens[3]	5
queens[4]	2
queens[5]	6
queens[6]	1
queens[7]	3

(a)



(b)

FIGURE 20.11 `queens[i]` denotes the position of the queen in row `i`.

Listing 20.10 gives the program that displays a solution for the Eight Queens problem.

LISTING 20.10 EightQueens.java

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class EightQueens extends JApplet {
5    public static final int SIZE = 8; // The size of the chessboard
6    private int[] queens = new int[SIZE]; // Queen positions
7
8    public EightQueens() {
9      search(0); // Search for a solution from row 0                    search for solution
10     add(new ChessBoard(), BorderLayout.CENTER); // Display solution
11   }
12
13   /** Check if a queen can be placed at row i and column j */
14   private boolean isValid(int row, int column) {                      check whether valid
15     for (int i = 1; i <= row; i++)
16       if (queens[row - i] == column // Check column
17         || queens[row - i] == column - i // Check upleft diagonal
18         || queens[row - i] == column + i) // Check upright diagonal
19         return false; // There is a conflict
20     return true; // No conflict
21   }
22
23   /** Search for a solution starting from a specified row */
24   private boolean search(int row) {                                   search this row
25     if (row == SIZE) // Stopping condition
26       return true; // A solution found to place 8 queens in 8 rows    search columns
27
28     for (int column = 0; column < SIZE; column++) {                   search next row
```

found

```
29        queens[row] = column; // Place a queen at (row, column)
30        if (isValid(row, column) && search(row + 1))
31          return true; // Found, thus return true to exit for loop
32      }
33
34      // No solution for a queen placed at any column of this row
35      return false;
36    }
37
38    class ChessBoard extends JPanel {
39      private Image queenImage =
40        new ImageIcon("image/queen.jpg").getImage();
41
42      ChessBoard() {
43        this.setBorder(BorderFactory.createLineBorder(Color.BLACK, 2));
44      }
45
46      protected void paintComponent(Graphics g) {
47        super.paintComponent(g);
48
49        // Paint the queens
50        for (int i = 0; i < SIZE; i++) {
51          int j = queens[i]; // The position of the queen in row i
52          g.drawImage(queenImage, j * getWidth() / SIZE,
53            i * getHeight() / SIZE, getWidth() / SIZE,
54            getHeight() / SIZE, this);
55        }
56
57        // Draw the horizontal and vertical lines
58        for (int i = 1; i < SIZE; i++) {
59          g.drawLine(0, i * getHeight() / SIZE,
60            getWidth(), i * getHeight() / SIZE);
61          g.drawLine(i * getWidth() / SIZE, 0,
62            i * getWidth() / SIZE, getHeight());
63        }
64      }
65    }
```

main method omitted

```
66 }
```

The program invokes `search(0)` (line 9) to start a search for a solution at row `0`, which recursively invokes `search(1)`, `search(2)`, . . . , and `search(7)` (line 30).

The recursive `search(row)` method returns `true` if all row are filled (lines 25–26). The method checks whether a queen can be placed in column `0`, `1`, `2`, . . . , and `7` in a `for` loop (line 28). Place a queen in the column (line 29). If the placement is valid, recursively search for the next row by invoking `search(row + 1)` (line 30). If search is successful, return `true` (line 31) to exit the `for` loop. In this case, there is no need to look for the next column in the row. If there is no solution for a queen to be placed on any column of this row, the method returns `false` (line 35).

Suppose you invoke `search(row)` for `row 3`, as shown in Figure 20.12(a). The method tries to fill in a queen in column `0`, `1`, `2`, and so on in this order. For each trial, the `isValid(row, column)` method (line 30) is called to check whether placing a queen at the specified position causes a conflict with the queens placed earlier. It ensures that no queen is placed in the same column (line 16), no queen is placed in the upper left diagonal (line 17), and no queen is placed in the upper right diagonal (line 18), as shown in Figure 20.12(a). If `isValid(row, column)` returns `false`, check the next column, as shown Figure 20.12(b). If `isValid(row, column)` returns `true`, recursively invoke `search(row + 1)`, as shown in Figure 20.12(d). If `search(row + 1)` returns `false`, check the next column on the preceding row, as shown Figure 20.12(c).

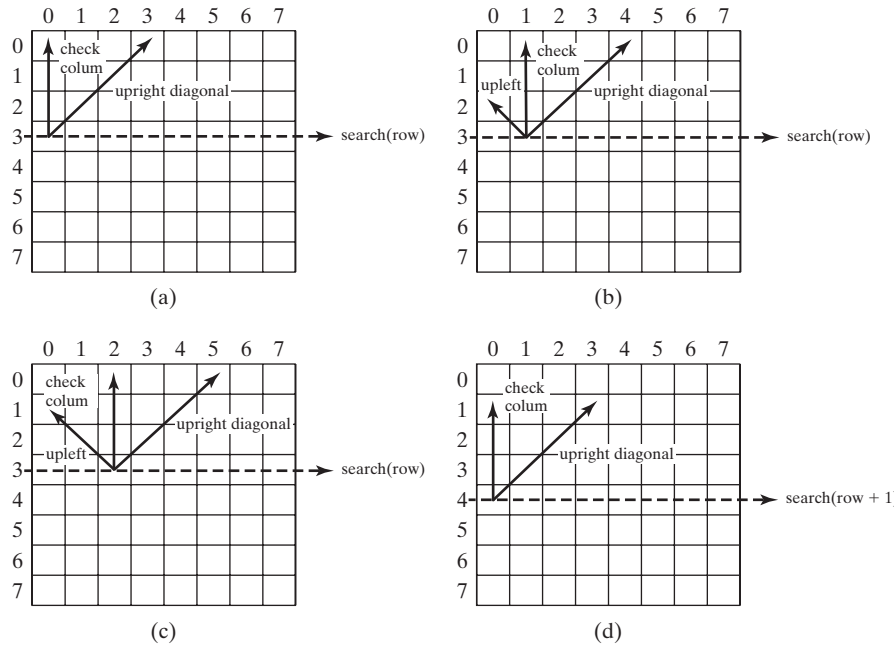


FIGURE 20.12 Invoking `search(row)` fills in a queen in a column on the row.

20.10 Recursion vs. Iteration

Recursion is an alternative form of program control. It is essentially repetition without a loop. When you use loops, you specify a loop body. The repetition of the loop body is controlled by the loop control structure. In recursion, the method itself is called repeatedly. A selection statement must be used to control whether to call the method recursively or not.

recursion overhead

Recursion bears substantial overhead. Each time the program calls a method, the system must assign space for all of the method's local variables and parameters. This can consume considerable memory and requires extra time to manage the additional space.

recursion advantages

Any problem that can be solved recursively can be solved nonrecursively with iterations. Recursion has some negative aspects: it uses up too much time and too much memory. Why, then, should you use it? In some cases, using recursion enables you to specify for an inherently recursive problem a clear, simple solution that would otherwise be difficult to obtain. Examples are the directory-size problem, the Towers of Hanoi problem, and the fractal problem, which are rather difficult to solve without using recursion.

recursion or iteration?

The decision whether to use recursion or iteration should be based on the nature of, and your understanding of, the problem you are trying to solve. The rule of thumb is to use whichever approach can best develop an intuitive solution that naturally mirrors the problem. If an iterative solution is obvious, use it. It will generally be more efficient than the recursive option.



Note

Your recursive program could run out of memory, causing a `StackOverflowError`.

`StackOverflowError`



Tip

If you are concerned about your program's performance, avoid using recursion, because it takes more time and consumes more memory than iteration.

performance concern

20.11 Tail Recursion

A recursive method is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call. For example, the recursive `isPalindrome` method

(lines 6–13) in Listing 20.4 is tail recursive because there are no pending operations after recursively invoking `isPalindrome` in line 12. However, the recursive `factorial` method (lines 16-21) in Listing 20.1 is not tail recursive, because there is a pending operation, namely multiplication, to be performed on return from each recursive call.

Tail recursion is desirable, because the method ends when the last recursive call ends. So there is no need to store the intermediate calls in the stack. Some compilers can optimize tail recursion to reduce stack space.

A non-tail-recursive method can often be converted to a tail-recursive method by using auxiliary parameters. These parameters are used to contain the result. The idea is to incorporate the pending operations into the auxiliary parameters in such a way that the recursive call no longer has a pending operation. You may define a new auxiliary recursive method with the auxiliary parameters. This method may overload the original method with the same name but a different signature. For example, the `factorial` method in Listing 20.1 can be written in a tail-recursive way as follows:

original method
invoke auxiliary method
auxiliary method
recursive call

```
1  /** Return the factorial for a specified number */
2  public static long factorial(int n) {
3    return factorial(n, 1); // Call auxiliary method
4  }
5
6  /** Auxiliary tail-recursive method for factorial */
7  private static long factorial(int n, int result) {
8    if (n == 1)
9      return result;
10   else
11     return factorial(n - 1, n * result); // Recursive call
12 }
```

The first `factorial` method simply invokes the second auxiliary method (line 3). The second method contains an auxiliary parameter `result` that stores the result for factorial of `n`. This method is invoked recursively in line 11. There is no pending operation after a call is returned. The final result is returned in line 9, which is also the return value from invoking `factorial(n, 1)` in line 3.

Key Terms

base case 678
infinite recursion 679
recursive method 678
recursive helper method 685
stopping condition 678
tail recursion 697

Chapter Summary

1. A recursive method is one that directly or indirectly invokes itself. For a recursive method to terminate, there must be one or more base cases.
2. Recursion is an alternative form of program control. It is essentially repetition without a loop control. It can be used to specify simple, clear solutions for inherently recursive problems that would otherwise be difficult to solve.
3. Sometimes the original method needs to be modified to receive additional parameters in order to be invoked recursively. A recursive helper method can be defined for this purpose.
4. Recursion bears substantial overhead. Each time the program calls a method, the system must assign space for all of the method's local variables and parameters. This can consume considerable memory and requires extra time to manage the additional space.
5. A recursive method is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call. Some compilers can optimize tail recursion to reduce stack space.

REVIEW QUESTIONS

Sections 20.1–20.3

20.1 What is a recursive method? Describe the characteristics of recursive methods. What is an infinite recursion?

20.2 Write a recursive mathematical definition for computing 2^n for a positive integer n.

20.3 Write a recursive mathematical definition for computing x^n for a positive integer n and a real number x.

20.4 Write a recursive mathematical definition for computing $1 + 2 + 3 + \cdots + n$ for a positive integer.

20.5 How many times is the `factorial` method in Listing 20.1 invoked for `factorial(6)`?

20.6 How many times is the `fib` method in Listing 20.2 invoked for `fib(6)`?

Sections 20.4–20.6

20.7 Show the call stack for `isPalindrome("abcba")` using the methods defined in Listing 20.3 and Listing 20.4, respectively.

20.8 Show the call stack for `selectionSort(new double[]{2, 3, 5, 1})` using the method defined in Listing 20.5.

20.9 What is a recursive helper method?

Section 20.7

20.10 How many times is the `moveDisks` method in Listing 20.8 invoked for `moveDisks(5, 'A', 'B', 'C')`?

Section 20.9

20.11 Which of the following statements are true?

- Any recursive method can be converted into a nonrecursive method.
- Recursive methods take more time and memory to execute than nonrecursive methods.
- Recursive methods are *always* simpler than nonrecursive methods.
- There is always a selection statement in a recursive method to check whether a base case is reached.

20.12 What is the cause for the stack-overflow exception?

Comprehensive

20.13 Show the output of the following program:

```
public class Test {
  public static void main(String[] args) {
    System.out.println(
      "Sum is " + xMethod(5));
  }

  public static int xMethod(int n) {
    if (n == 1)
      return 1;
    else
      return n + xMethod(n - 1);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    xMethod(1234567);
  }

  public static void xMethod(int n) {
    if (n > 0) {
      System.out.print(n % 10);
      xMethod(n / 10);
    }
  }
}
```

20.14 Show the output of the following two programs:

```
public class Test {
  public static void main(String[] args) {
    xMethod(5);
  }

  public static void xMethod(int n) {
    if (n > 0) {
      System.out.print(n + " ");
      xMethod(n - 1);
    }
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    xMethod(5);
  }

  public static void xMethod(int n) {
    if (n > 0) {
      xMethod(n - 1);
      System.out.print(n + " ");
    }
  }
}
```

20.15 What is wrong in the following method?

```
public class Test {
  public static void main(String[] args) {
    xMethod(1234567);
  }

  public static void xMethod(double n) {
    if (n != 0) {
      System.out.print(n);
      xMethod(n / 10);
    }
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    Test test = new Test();
    System.out.println(test.toString());
  }

  public Test() {
    Test test = new Test();
  }
}
```

20.16 Identify tail-recursive methods in this chapter.

20.17 Rewrite the `fib` method in Listing 20.2 using tail recursion.

Programming Exercises

Sections 20.2–20.3

20.1* (*Factorial*) Using the `BigInteger` class introduced in §14.12, you can find the factorial for a large number (e.g., `100!`). Write a program that prompts the user to enter an integer and displays its factorial. Implement the method using recursion.

20.2* (*Fibonacci numbers*) Rewrite the `fib` method in Listing 20.2 using iterations.

Hint: To compute `fib(n)` without recursion, you need to obtain `fib(n - 2)` and `fib(n - 1)` first. Let `f0` and `f1` denote the two previous Fibonacci numbers. The current Fibonacci number would then be `f0 + f1`. The algorithm can be described as follows:

```
f0 = 0; // For fib(0)
f1 = 1; // For fib(1)

for (int i = 1; i <= n; i++) {
  currentFib = f0 + f1;
  f0 = f1;
  f1 = currentFib;
}

// After the loop, currentFib is fib(n)
```

20.3* (*Computing greatest common divisor using recursion*) The `gcd(m, n)` can also be defined recursively as follows:

- If `m % n` is `0`, `gcd (m, n)` is `n`.
- Otherwise, `gcd(m, n)` is `gcd(n, m % n)`.

Write a recursive method to find the GCD. Write a test program that computes `gcd(24, 16)` and `gcd(255, 25)`.

20.4 (*Summing series*) Write a recursive method to compute the following series:

$$m(i) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{i}$$

20.5 (*Summing series*) Write a recursive method to compute the following series:

$$m(i) = \frac{1}{3} + \frac{2}{5} + \frac{3}{7} + \frac{4}{9} + \frac{5}{11} + \frac{6}{13} + \cdots + \frac{i}{2i+1}$$

20.6* (*Summing series*) Write a recursive method to compute the following series:

$$m(i) = \frac{1}{2} + \frac{2}{3} + \cdots + \frac{i}{i+1}$$

20.7* (*Fibonacci series*) Modify Listing 20.2, ComputeFibonacci.java, so that the program finds the number of times the `fib` method is called.

(*Hint*: Use a static variable and increment it every time the method is called.)

Section 20.4

20.8* (*Printing the digits in an integer reversely*) Write a recursive method that displays an `int` value reversely on the console using the following header:

```
public static void reverseDisplay(int value)
```

For example, `reverseDisplay(12345)` displays `54321`.

20.9* (*Printing the characters in a string reversely*) Write a recursive method that displays a string reversely on the console using the following header:

```
public static void reverseDisplay(String value)
```

For example, `reverseDisplay("abcd")` displays `dcba`.

20.10* (*Occurrences of a specified character in a string*) Write a recursive method that finds the number of occurrences of a specified letter in a string using the following method header.

```
public static int count(String str, char a)
```

For example, `count("Welcome", 'e')` returns `2`.

20.11* (*Summing the digits in an integer using recursion*) Write a recursive method that computes the sum of the digits in an integer. Use the following method header:

```
public static int sumDigits(long n)
```

For example, `sumDigits(234)` returns 2 + 3 + 4 = 9.

Section 20.5

20.12** (*Printing the characters in a string reversely*) Rewrite Exercise 20.9 using a helper method to pass the substring high index to the method. The helper method header is:

```
public static void reverseDisplay(String value, int high)
```

20.13* (*Finding the largest number in an array*) Write a recursive method that returns the largest integer in an array.

20.14* (*Finding the number of uppercase letters in a string*) Write a recursive method to return the number of uppercase letters in a string.

20.15* (*Occurrences of a specified character in a string*) Rewrite Exercise 20.10 using a helper method to pass the substring high index to the method. The helper method header is:

```
public static int count(String str, char a, int high)
```

20.16* (*Finding the number of uppercase letters in an array*) Write a recursive method to return the number of uppercase letters in an array of characters. You need to define the following two methods. The second one is a recursive helper method.

```
public static int count(char[] chars)
public static int count(char[] chars, int high)
```

20.17* (*Occurrences of a specified character in an array*) Write a recursive method that finds the number of occurrences of a specified character in an array. You need to define the following two methods. The second one is a recursive helper method.

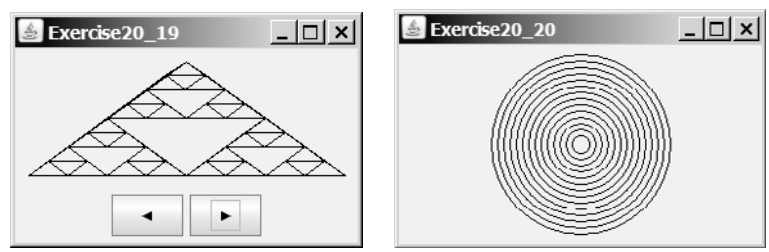
```
public static int count(char[] chars, char ch)
public static int count(char[] chars, char ch, int high)
```

Sections 20.6

20.18* (*Towers of Hanoi*) Modify Listing 20.8, TowersOfHanoi.java, so that the program finds the number of moves needed to move *n* disks from tower A to tower B.

(*Hint*: Use a static variable and increment it every time the method is called.)

20.19* (*Sierpinski triangle*) Revise Listing 20.9 to develop an applet that lets the user use the *Increase* and *Decrease* buttons to increase or decrease the current order by 1, as shown in Figure 20.13(a). The initial order is 0. If the current order is 0, the *Decrease* button is ignored.



(a) (b)

FIGURE 20.13 (a) Exercise 20.19 uses the *Increase* and *Decrease* buttons to increase or decrease the current order by 1. (b) Exercise 20.20 draws ovals using a recursive method.

20.20* (*Displaying circles*) Write a Java applet that displays ovals, as shown in Figure 20.13(b). The ovals are centered in the panel. The gap between two adjacent ovals is 10 pixels, and the gap between the panel and the largest oval is also 10.

Comprehensive

20.21* (*Decimal to binary*) Write a recursive method that converts a decimal number into a binary number as a string. The method header is as follows:

```
public static String decimalToBinary(int value)
```

20.22* (*Decimal to hex*) Write a recursive method that converts a decimal number into a hex number as a string. The method header is as follows:

```
public static String decimalToHex(int value)
```

20.23* (*Binary to decimal*) Write a recursive method that parses a binary number as a string into a decimal integer. The method header is as follows:

```
public static int binaryToDecimal(String binaryString)
```

20.24* (*Hex to decimal*) Write a recursive method that parses a hex number as a string into a decimal integer. The method header is as follows:

```
public static int hexToDecimal(String hexString)
```

20.25** (*String permutation*) Write a recursive method to print all the permutations of a string. For example, for a string `abc`, the printout is

```
abc
acb
bac
bca
cab
cba
```

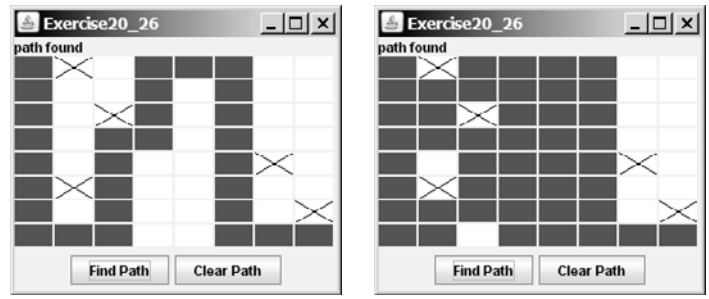
(*Hint*: Define the following two methods. The second is a helper method.)

```
public static void displayPermuation(String s)
public static void displayPermuation(String s1, String s2)
```

The first method simply invokes `displayPermuation("", s)`. The second method uses a loop to move a character from `s2` to `s1` and recursively invoke it with a new `s1` and `s2`. The base case is that `s2` is empty and prints `s1` to the console.

20.26** (*Creating a maze*) Write an applet that will find a path in a maze, as shown in Figure 20.14(a). The maze is represented by an 8×8 board. The path must meet the following conditions:

- The path is between the upper-left corner cell and the lower-right corner cell in the maze.
- The applet enables the user to place or remove a mark on a cell. A path consists of adjacent unmarked cells. Two cells are said to be adjacent if they are horizontal or vertical neighbors, but not if they are diagonal neighbors.
- The path does not contain cells that form a square. The path in Figure 20.14(b), for example, does not meet this condition. (The condition makes a path easy to identify on the board.)

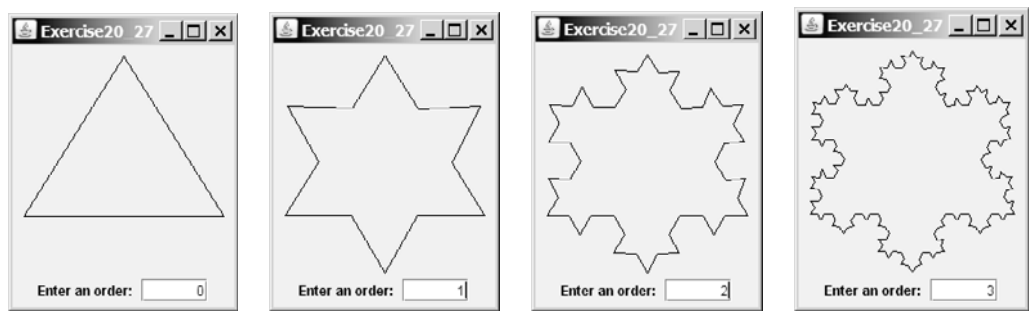


(a) Correct path (b) Illegal path

FIGURE 20.14 The program finds a path from the upper-left corner to the bottom-right corner.

20.27** (*Koch snowflake fractal*) The text presented the Sierpinski triangle fractal. In this exercise, you will write an applet to display another fractal, called the *Koch snowflake*, named after a famous Swedish mathematician. A Koch snowflake is created as follows:

1. Begin with an equilateral triangle, which is considered to be the Koch fractal of order (or level) **0**, as shown in Figure 20.15(a).
2. Divide each line in the shape into three equal line segments and draw an outward equilateral triangle with the middle line segment as the base to create a Koch fractal of order **1**, as shown in Figure 20.15(b).
3. Repeat step 2 to create a Koch fractal of order **2**, **3**, . . . , and so on, as shown in Figure 20.15(c–d).



(a) (b) (c) (d)

FIGURE 20.15 A Koch snowflake is a fractal starting with a triangle.

20.28** (*Nonrecursive directory size*) Rewrite Listing 20.7, DirectorySize.java, without using recursion.

20.29* (*Number of files in a directory*) Write a program that prompts the user to enter a directory and displays the number of the files in the directory.

20.30** (*Finding words*) Write a program that finds all occurrences of a word in all the files under a directory, recursively. Pass the parameters from the command line as follows:

```
java Exercise20_30 dirName word
```



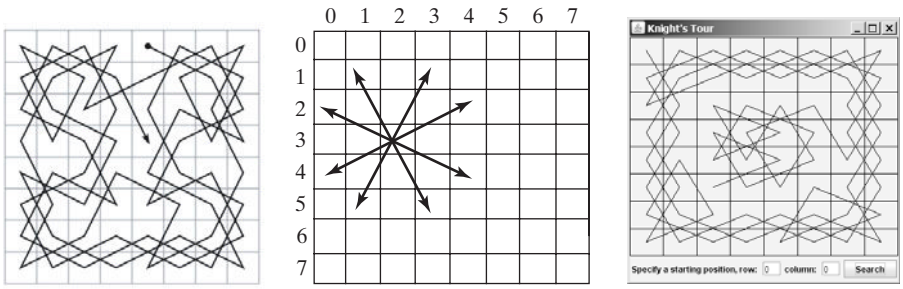
Video Note
Search a string in a directory

20.31** (*Replacing words*) Write a program that replaces all occurrences of a word with a new word in all the files under a directory, recursively. Pass the parameters from the command line as follows:

```
java Exercise20_31 dirName oldWord newWord
```

20.32* (*Game: Knight's Tour*) The Knight's Tour is an ancient puzzle. The objective is to move a knight, starting from any square on a chessboard, to every other square once, as shown in Figure 20.16(a). Note that the knight makes only L-shape moves (two spaces in one direction and one space in a perpendicular direction). As shown in Figure 20.16(b), the knight can move to eight squares. Write a program that displays the moves for the knight in an applet, as shown in Figure 20.16(c).

(*Hint*: A brute-force approach for this problem is to move the knight from one square to another available square arbitrarily. Using such an approach, your program will take a long time to finish. A better approach is to employ some heuristics. A knight has two, three, four, six, or eight possible moves, depending on its location. Intuitively, you should attempt to move the knight to the least accessible squares first and leave those more accessible squares open, so there will be a better chance of success at the end of the search.)



(a) (b) (c)

FIGURE 20.16 (a) A knight traverses all squares once. (b) A knight makes an L-shape move. (c) An applet displays a knight tour path.

20.33* (*Game: Knight's Tour animation*) Write an applet for the Knight's Tour problem. Your applet should let the user move a knight to any starting square and click the *Solve* button to animate a knight moving along the path, as shown in Figure 20.17.

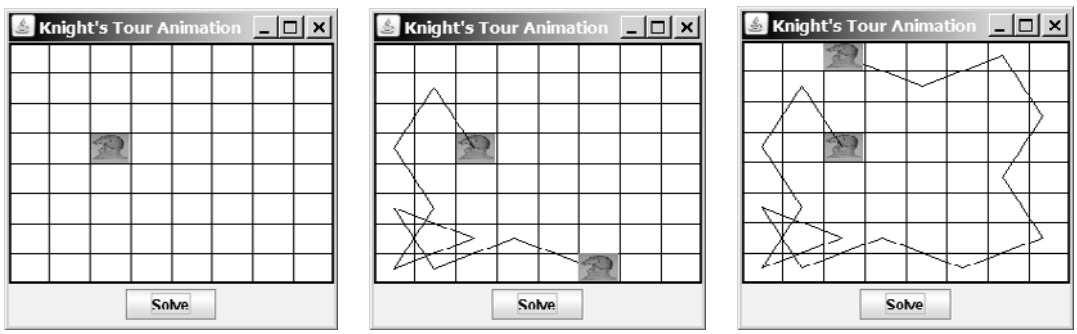
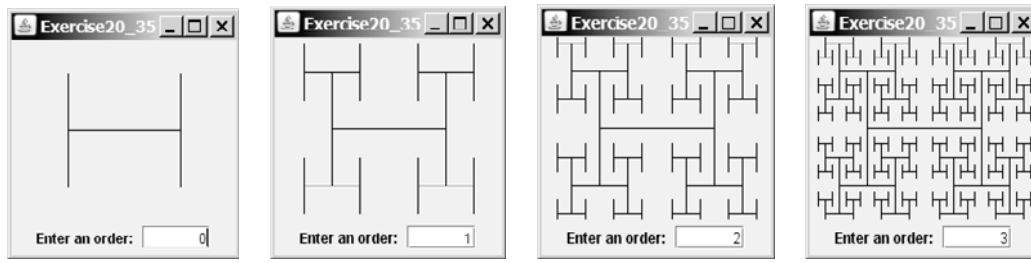


FIGURE 20.17 A knight traverses along the path.

20.34** (*Game: Sudoku*) Write a program to solve the Sudoku problem using recursion.

20.35** (*H-tree fractal*) An H-tree is a fractal defined as follows:

1. Begin with a letter H. The three lines of the H are of the same length, as shown in Figure 20.18(a).
2. The letter H (in its sans-serif form, H) has four endpoints. Draw an H centered at each of the four endpoints to an H-tree of order 1, as shown in Figure 20.18(b). These H's are half the size of the H that contains the four endpoints.
3. Repeat step 2 to create a H-tree of order 2, 3, . . . , and so on, as shown in Figure 20.18(c–d).



(a) (b) (c) (d)

FIGURE 20.18 An H-tree is a fractal starting with three lines of equal length in an H-shape.

The H-tree is used in VLSI design as a clock distribution network for routing timing signals to all parts of a chip with equal propagation delays. Write an applet that draws an H-tree, as shown in Figure 20.18.

20.36* (*Game: all Sudoku solutions*) Rewrite Exercise 20.34 to find all possible solutions to a Sudoku problem.

20.37* (*Game: multiple Eight Queens solution*) Write an applet to display all possible solutions for the Eight Queens puzzle in a scroll pane, as shown in Figure 20.19. For each solution, put a label to denote the solution number.

Hint: Place all solution panels into one panel and place this one panel into a `JScrollPane`. The solution panel class should override the `getPreferredSize()` method to ensure that a solution panel is displayed properly. See Listing 15.3, FigurePanel.java, on how to override `getPreferredSize()`.

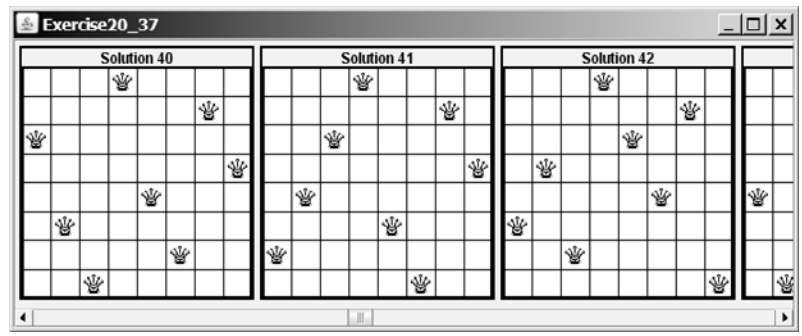
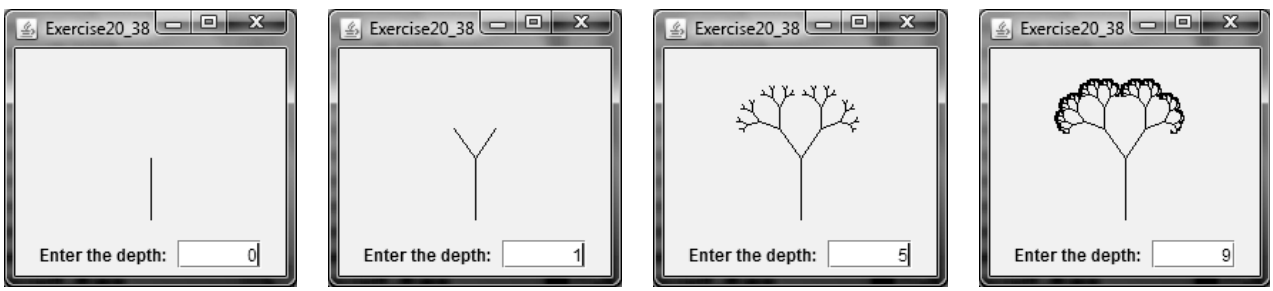


FIGURE 20.19 All solutions are placed in a scroll pane.

20.38** (*Recursive tree*) Write an applet to display a recursive tree as shown in Figure 20.20.

20.39** (*Dragging the tree*) Revise Exercise 20.38 to move the tree to where the mouse is dragged.



(a) (b) (c) (d)

FIGURE 20.20 A recursive tree with the specified depth.

APPENDIXES

Appendix A
Java Keywords

Appendix B
The ASCII Character Set

Appendix C
Operator Precedence Chart

Appendix D
Java Modifiers

Appendix E
Special Floating-Point Values

Appendix F
Number Systems

This page intentionally left blank

APPENDIX A

Java Keywords

The following fifty keywords are reserved for use by the Java language:

`abstract`	`double`	`int`	`super`
`assert`	`else`	`interface`	`switch`
`boolean`	`enum`	`long`	`synchronized`
`break`	`extends`	`native`	`this`
`byte`	`for`	`new`	`throw`
`case`	`final`	`package`	`throws`
`catch`	`finally`	`private`	`transient`
`char`	`float`	`protected`	`try`
`class`	`goto`	`public`	`void`
`const`	`if`	`return`	`volatile`
`continue`	`implements`	`short`	`while`
`default`	`import`	`static`	
`do`	`instanceof`	`strictfp*`	

The keywords **`goto`** and **`const`** are C++ keywords reserved, but not currently used, in Java. This enables Java compilers to identify them and to produce better error messages if they appear in Java programs.

The literal values **`true`**, **`false`**, and **`null`** are not keywords, just like literal value **`100`**. However, you cannot use them as identifiers, just as you cannot use **`100`** as an identifier.

`assert` is a keyword added in JDK 1.4 and **`enum`** is a keyword added in JDK 1.5.

*The **`strictfp`** keyword is a modifier for method or class to use strict floating-point calculations. Floating-point arithmetic can be executed in one of two modes: *strict* or *nonstrict*. The strict mode guarantees that the evaluation result is the same on all Java Virtual Machine implementations. The nonstrict mode allows intermediate results from calculations to be stored in an extended format different from the standard IEEE floating-point number format. The extended format is machine-dependent and enables code to be executed faster. However, when you execute the code using the nonstrict mode on different JVMs, you may not always get precisely the same results. By default, the nonstrict mode is used for floating-point calculations. To use the strict mode in a method or a class, add the **`strictfp`** keyword in the method or the class declaration. Strict floating-point may give you slightly better precision than nonstrict floating-point, but the distinction will only affect some applications. Strictness is not inherited; that is, the presence of **`strictfp`** on a class or interface declaration does not cause extended classes or interfaces to be strict.

APPENDIX B

The ASCII Character Set

Tables B.1 and B.2 show ASCII characters and their respective decimal and hexadecimal codes. The decimal or hexadecimal code of a character is a combination of its row index and column index. For example, in Table B.1, the letter A is at row 6 and column 5, so its decimal equivalent is 65; in Table B.2, letter A is at row 4 and column 1, so its hexadecimal equivalent is 41.

TABLE B.1 ASCII Character Set in the Decimal Index

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dcl	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	”	#	$	%	&	’
4	(	)	*	+	,	−	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	−	‘	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

TABLE B.2 ASCII Character Set in the Hexadecimal Index

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht	nl	vt	ff	cr	so	si
1	dle	dcl	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
2	sp	!	”	#	$	%	&	’	(	)	*	+	,	−	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	−
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	del

APPENDIX C

Operator Precedence Chart

The operators are shown in decreasing order of precedence from top to bottom. Operators in the same group have the same precedence, and their associativity is shown in the table.

Operator	*Name*	*Associativity*
`()`	Parentheses	Left to right
`()`	Function call	Left to right
`[]`	Array subscript	Left to right
`.`	Object member access	Left to right
`++`	Postincrement	Right to left
`--`	Postdecrement	Right to left
`++`	Preincrement	Right to left
`--`	Predecrement	Right to left
`+`	Unary plus	Right to left
`-`	Unary minus	Right to left
`!`	Unary logical negation	Right to left
`(type)`	Unary casting	Right to left
`new`	Creating object	Right to left
`*`	Multiplication	Left to right
`/`	Division	Left to right
`%`	Remainder	Left to right
`+`	Addition	Left to right
`-`	Subtraction	Left to right
`<<`	Left shift	Left to right
`>>`	Right shift with sign extension	Left to right
`>>>`	Right shift with zero extension	Left to right
`<`	Less than	Left to right
`<=`	Less than or equal to	Left to right
`>`	Greater than	Left to right
`>=`	Greater than or equal to	Left to right
`instanceof`	Checking object type	Left to right

Operator	*Name*	*Associativity*
`==`	Equal comparison	Left to right
`!=`	Not equal	Left to right
`&`	(Unconditional AND)	Left to right
`^`	(Exclusive OR)	Left to right
`|`	(Unconditional OR)	Left to right
`&&`	Conditional AND	Left to right
`||`	Conditional OR	Left to right
`?:`	Ternary condition	Right to left
`=`	Assignment	Right to left
`+=`	Addition assignment	Right to left
`-=`	Subtraction assignment	Right to left
`*=`	Multiplication assignment	Right to left
`/=`	Division assignment	Right to left
`%=`	Remainder assignment	Right to left

APPENDIX D

Java Modifiers

Modifiers are used on classes and class members (constructors, methods, data, and class-level blocks), but the final modifier can also be used on local variables in a method. A modifier that can be applied to a class is called a *class modifier*. A modifier that can be applied to a method is called a *method modifier*. A modifier that can be applied to a data field is called a *data modifier*. A modifier that can be applied to a class-level block is called a block modifier. The following table gives a summary of the Java modifiers.

Modifier	*class*	*constructor*	*method*	*data*	*block*	*Explanation*
`(default)*`	√	√	√	√	√	A class, constructor, method, or data field is visible in this package.
`public`	√	√	√	√		A class, constructor, method, or data field is visible to all the programs in any package.
`private`		√	√	√		A constructor, method or data field is only visible in this class.
`protected`		√	√	√		A constructor, method or data field is visible in this package and in subclasses of this class in any package.
`static`			√	√	√	Define a class method, or a class data field or a static initialization block.
`final`	√		√	√		A final class cannot be extended. A final method cannot be modified in a subclass. A final data field is a constant.
`abstract`	√		√			An abstract class must be extended. An abstract method must be implemented in a concrete subclass.
`native`			√			A native method indicates that the method is implemented using a language other than Java.

*Default access has no modifier associated with it. For example: `class Test {}`

Modifier	*class*	*constructor*	*method*	*data*	*block*	*Explanation*
`synchronized`			√		√	Only one thread at a time can execute this method.
`strictfp`	√		√			Use strict floating-point calculations to guarantee that the evaluation result is the same on all JVMs.
`transient`				√		Mark a nonserializable instance data field.

APPENDIX E

Special Floating-Point Values

Dividing an integer by zero is invalid and throws `ArithmeticException`, but dividing a floating-point value by zero does not cause an exception. Floating-point arithmetic can overflow to infinity if the result of the operation is too large for a `double` or a `float`, or underflow to zero if the result is too small for a double or a `float`. Java provides the special floating-point values `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, and `NaN` (Not a Number) to denote these results. These values are defined as special constants in the `Float` class and the Double class.

If a positive floating-point number is divided by zero, the result is `POSITIVE_INFINITY`. If a negative floating-point number is divided by zero, the result is `NEGATIVE_INFINITY`. If a floating-point zero is divided by zero, the result is NaN, which means that the result is undefined mathematically. The string representation of these three values are Infinity, -Infinity, and NaN. For example,

```
System.out.print(1.0 / 0); // Print Infinity
System.out.print(-1.0 / 0); // Print -Infinity
System.out.print(0.0 / 0); // Print NaN
```

These special values can also be used as operands in computations. For example, a number divided by `POSITIVE_INFINITY` yields a positive zero. Table E.1 summarizes various combinations of the `/`, `*`, `%`, `+`, and `-` operators.

TABLE E.1 Special Floating-Point Values

x	*y*	*x*/*y*	*x***y*	*x*%*y*	*x* + *y*	*x* − *y*
Finite	± 0.0	± ∞	± 0.0	NaN	Finite	Finite
Finite	± ∞	± 0.0	± 0.0	x	± ∞	∞
± 0.0	± 0.0	NaN	± 0.0	NaN	± 0.0	± 0.0
± ∞	Finite	± ∞	± 0.0	NaN	± ∞	± ∞
± ∞	± ∞	NaN	± 0.0	NaN	± ∞	∞
± 0.0	± ∞	± 0.0	NaN	± 0.0	± ∞	± 0.0
NaN	Any	NaN	NaN	NaN	NaN	NaN
Any	NaN	NaN	NaN	NaN	NaN	NaN



Note

If one of the operands is NaN, the result is NaN.

APPENDIX F

Number Systems

1 Introduction

binary numbers

Computers use binary numbers internally, because computers are made naturally to store and process 0s and 1s. The binary number system has two digits, 0 and 1. A number or character is stored as a sequence of 0s and 1s. Each 0 or 1 is called a *bit* (binary digit).

decimal numbers

In our daily life we use decimal numbers. When we write a number such as 20 in a program, it is assumed to be a decimal number. Internally, computer software is used to convert decimal numbers into binary numbers, and vice versa.

We write computer programs using decimal numbers. However, to deal with an operating system, we need to reach down to the "machine level" by using binary numbers. Binary numbers tend to be very long and cumbersome. Often hexadecimal numbers are used to abbreviate them, with each hexadecimal digit representing four binary digits. The hexadecimal number system has 16 digits: 0–9, A–F. The letters A, B, C, D, E, and F correspond to the decimal numbers 10, 11, 12, 13, 14, and 15.

hexadecimal number

The digits in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. A decimal number is represented by a sequence of one or more of these digits. The value that each digit represents depends on its position, which denotes an integral power of 10. For example, the digits 7, 4, 2, and 3 in decimal number 7423 represent 7000, 400, 20, and 3, respectively, as shown below:

7	4	2	3

$$= 7 \times 10^3 + 4 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

$$10^3\ 10^2\ 10^1\ 10^0 = 7000 + 400 + 20 + 3 = 7423$$

base
radix

The decimal number system has ten digits, and the position values are integral powers of 10. We say that 10 is the *base* or *radix* of the decimal number system. Similarly, since the binary number system has two digits, its base is 2, and since the hex number system has 16 digits, its base is 16.

If 1101 is a binary number, the digits 1, 1, 0, and 1 represent 1×2^3, 1×2^2, 0×2^1, and 1×2^0, respectively:

1	1	0	1

$$= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$2^3\ \ 2^2\ \ 2^1\ \ 2^0 = 8 + 4 + 0 + 1 = 13$$

If 7423 is a hex number, the digits 7, 4, 2, and 3 represent 7×16^3, 4×16^2, 2×16^1, and 3×16^0, respectively:

7	4	2	3

$$= 7 \times 16^3 + 4 \times 16^2 + 2 \times 16^1 + 3 \times 16^0$$

$$16^3\ 16^2\ 16^1\ 16^0 = 28672 + 1024 + 32 + 3 = 29731$$

2 Conversions Between Binary and Decimal Numbers

binary to decimal

Given a binary number $b_n b_{n-1} b_{n-2} \ldots b_2 b_1 b_0$, the equivalent decimal value is

$$b_n \times 2^n + b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \ldots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

Here are some examples of converting binary numbers to decimals:

Binary	Conversion Formula	Decimal
10	$1 \times 2^1 + 0 \times 2^0$	2
1000	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$	8
10101011	$1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	171

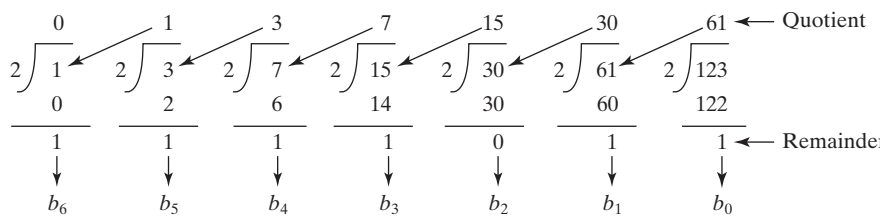
decimal to binary

To convert a decimal number d to a binary number is to find the bits b_n, b_{n-1}, b_{n-2}, ..., b_2, b_1, and b_0 such that

$$d = b_n \times 2^n + b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \ldots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

These bits can be found by successively dividing d by 2 until the quotient is 0. The remainders are b_0, b_1, b_2, ..., b_{n-2}, b_{n-1}, and b_n.

For example, the decimal number 123 is 1111011 in binary. The conversion is done as follows:





Tip

The Windows Calculator, as shown in Figure F.1, is a useful tool for performing number conversions. To run it, choose *Programs*, *Accessories*, and *Calculator* from the *Start* button, then under *View* select *Scientific*.

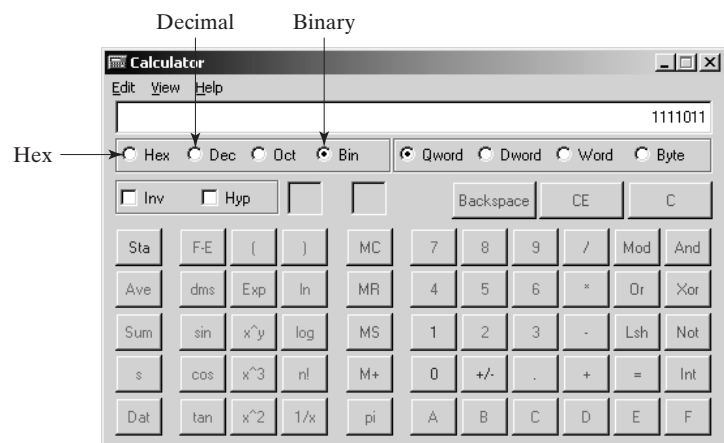

Decimal
Binary
Hex
Calculator
Edit View Help
1111011
Hex Dec Oct Bin
Qword Dword Word Byte
Inv Hyp
Backspace CE C
Sta F-E ( ) MC 7 8 9 / Mod And
Ave dms Exp ln MR 4 5 6 * Or Xor
Sum sin x^y log MS 1 2 3 - Lsh Not
s cos x^3 n! M+ 0 +/- . + = Int
Dat tan x^2 1/x pi A B C D E F

FIGURE F.1 You can perform number conversions using the Windows Calculator.

3 Conversions Between Hexadecimal and Decimal Numbers

Given a hexadecimal number $h_n h_{n-1} h_{n-2} \ldots h_2 h_1 h_0$, the equivalent decimal value is

hex to decimal

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

Here are some examples of converting hexadecimal numbers to decimals:

Hexadecimal	Conversion Formula	Decimal
7F	$7 \times 16^1 + 15 \times 16^0$	127
FFFF	$15 \times 16^3 + 15 \times 16^2 + 15 \times 16^1 + 15 \times 16^0$	65535
431	$4 \times 16^2 + 3 \times 16^1 + 1 \times 16^0$	1073

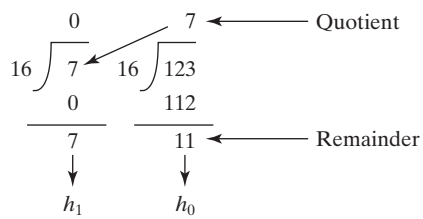
To convert a decimal number d to a hexadecimal number is to find the hexadecimal digits $h_n, h_{n-1}, h_{n-2}, \ldots, h_2, h_1$, and h_0 such that

decimal to hex

$$d = h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

These numbers can be found by successively dividing d by 16 until the quotient is 0. The remainders are $h_0, h_1, h_2, \ldots, h_{n-2}, h_{n-1}$, and h_n.

For example, the decimal number 123 is 7B in hexadecimal. The conversion is done as follows:



4 Conversions Between Binary and Hexadecimal Numbers

To convert a hexadecimal to a binary number, simply convert each digit in the hexadecimal number into a four-digit binary number, using Table F.1.

hex to binary

For example, the hexadecimal number 7B is 1111011, where 7 is 111 in binary, and B is 1011 in binary.

To convert a binary number to a hexadecimal, convert every four binary digits from right to left in the binary number into a hexadecimal number.

binary to hex

For example, the binary number 1110001101 is 38D, since 1101 is D, 1000 is 8, and 11 is 3, as shown below.

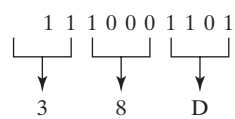


TABLE F.1 Converting Hexadecimal to Binary

Hexadecimal	*Binary*	*Decimal*
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

(*Note*: Octal numbers are also useful. The octal number system has eight digits, 0 to 7. A decimal number 8 is represented in the octal system as 10.)

REVIEW QUESTIONS

1. Convert the following decimal numbers into hexadecimal and binary numbers.

   `100; 4340; 2000`

2. Convert the following binary numbers into hexadecimal and decimal numbers.

   `1000011001; 100000000; 100111`

3. Convert the following hexadecimal numbers into binary and decimal numbers.

   `FEFA9; 93; 2000`

INDEX

- (subtraction), 33
-= (subtraction assignment operator), 39
--var (predecrement), 40
!= (not equal to), 72
! operator, truth table for, 88
&& operator, truth table for, 88
$ character, 29
% (remainder), 33
%= (remainder assignment operator), 37
* character (all), 322
* (multiplication), 33
*= (multiplication assignment operator), 37
/ (division), 33
/= (division assignment operator), 37
\ (backslash), 46, 323
^ operator, truth table for, 88
|| operator, truth table for, 88
+ (addition), 33
+= (addition assignment operator), 39
++var (preincrement), 40
var++ (postincrement), 40
< (less than), 72
<= (less than or equal to), 72
<applet> tag, syntax of, 615–617
<param> tag, 616
== (equal to), 72
> (greater than), 72
>= (greater than or equal to), 72
var-- (postdecrement), 40

A

abs method, Math class, 174
Absolute file name, 323–324
Abstract classes, 458–462
  Calendar class, 462–465
    field constants in, 463
    TestCalendar.java, 463–465
  Circle.java, 460
  defined, 458
  GeometricObject.java, 458–460
  GregorianCalendar class, 462–463
  interesting points on, 462
  interfaces vs., 465–467
  Rectangle.java, 460
Abstract methods, 458–459, 460, 471, 474, 478, 486
abstract modifier, 458
Abstract Windows Toolkit (AWT), Swing vs., 406
AbstractButton class, 572–573, 575, 578, 581, 664
Accessibility modifiers, 395
Accessible method, overriding, 383–384
action-after-each-iteration, 126–128, 136
ActionEvent, 470–471, 534–538, 557–560, 575–586, 590, 592, 626
ActionListener interface, 469–471, 534, 536
Actual parameter, 157, 320
AdapterDemo.java, 551–552
addActionListener method, 536
Addition (+), 33
Addition assignment operator (+=), 39
AdditionQuiz.java, 73
AdjustmentEvent, 487
AdjustmentListener interface, 596
Aggregated class, 354
Aggregated objects (subject object), 354
Aggregating class, 354
Aggregating object (owner object), 354
Aggregation, 353–355
Aggregation relationship, representation of, 353–354
Algorithms, 24, 216, 219, 242
alignment property, FlowLayout manager, 411–471, 505
Alignments, 574
alt attribute, 562
Animal class, 465, 475
AnimationDemo.java, 558–559
Anonymous arrays, 209
Anonymous class listeners, 542–544
Anonymous objects, 272
AnonymousListenerDemo.java, 543–544
  enlarging or shrinking a circle (example), 539–541
Applet class, 618–620
  destroy method, 620
  getParameter method, 621
  init method, 620
  start method, 620
  stop method, 620
  subclasses of, 618
Applets, 12–13, 15
  audio, playing in Java programs, 633–634
  bouncing ball (case study), 644
  demos, 615
  enabling to run as applications, 618–619
  HTML file and the tag, 615–618
  JApplet class, 618
  and multimedia, 613–637
  multimedia animations, 634–637
  passing strings to, 620–624
  TicTacToe game (case study), 628–632
  viewing from a Web browser, 617
  viewing using the applet, 617–618
  viewer utility, 617–618
appletviewer command, 617–618
Application program interface (API), 10
arcAngle, 499, 506
archive attribute, 616
Arcs, drawing, 506–507
  DrawArcs.java, 506–507
Arguments, 143 (60 entries)
Arithmetic/logic unit (ALU), 2–3
ArithmeticException, 432–435, 437–438, 482, 716
Array elements, analyzing (problem), 185–186
  grade assignment (problem), 186–187
  TestArray.java, 390–393
Array initializers, 201
Array lists, 390
Array objects, 271
Array variables:
  arrays vs., 200
  declaring, 199
arraycopy method, 208–209
ArrayIndexOutOfBoundsException, 203, 432, 443
ArrayList class, 390–393
  add(Object) method, 391
  differences and similarities between arrays and, 392
  remove(index) method, 391
  remove(Object) method, 391
  size() method, 391
  TestArrayList.java, 392
  toString method, 392
Arrays, 197–223
  anonymous, 209
  array initializers, 201
  array length, 200
  array variables, declaring, 199
  Arrays class, 223
  basics of, 198–199
  binary search approach, 217–219
  conversion between strings and, 309
  copying, 208–209
  creating, 199–200
  defined, 198
  insertion sort, 220–222
  linear search approach, 216–217
  multidimensional198
  of objects, 232
  passing to methods, 209–212
  processing, 201–203
  returning from a method, 212–215
  searching, 216–219
  selection sort, 219–220
  serializing, 665
  sorting, 219–222
  GenericSort.java, 479–481
  two-dimensional, 235
  variable-length argument lists, 215–216
Arrays class, 223
  binarySearch method, 217–219
  sort method, 219–222
Arrow keys, 5
Ascent, 510
ASCII (American Standard Code for Information Interchange), 45
ASCII character set, 710–711
Assembler, 6
Assembly language, 5–6,
Assignment expression, 30–31

Assignment operator, 30–31
Assignment statement, 31
Association, 162
Associativity, 97–98
Audio:
  ImageAudioAnimation.java, 635–637
  playing in Java programs, 634

B

Backslash (\), 323
BallControl.java, 626–627
Ball.java, 625–626
Base, decimal number system, 717
BASIC, 6
Behavior, of objects, 264
BigDecimal class, 488
BigInteger class, 488
Binary files, 650, 652
Binary I/O, 649–668
  characters/strings in, 656–657
  copying files (problem), 660–662
  DataInputStream class, 655–657
  DataOutputStream class, 655–657
  object I/O, 662–666
  random access files, 666–668
  Serializable interface, 664–665
  TestFileStream.java, 654–655
  text I/O vs., 650–652
Binary numbers, 3, 9
  conversions between decimal numbers and, 718
  conversions between hexadecimal numbers and, 719–720
Binary operator, 35, 97
Binary search, 1 217
  binarySearch method, 217–219
  BinarySearch.java, 218–219
  recursive binary search method, 686–687
Birth dates, guessing (problem), 250–251
  GuessBirthDateUsingArray.java, 250–251
Bits, 3, 9
Block, 16
Block comment, 16–17
Block styles, 52–53
BMI class, 351–353
  BMI.java, 352–353
  getBMI() instance method, 353
  getStatus() instance method, 353
  UseBMIClass.java, 352–353
Body Mass Index (BMI), computing (problem), 84–85
Boolean literal, 73, 272
Boolean operators, 88
Boolean variables, 72
Boolean wrapper class, 477
BorderLayout manager, 415–417
  ShowBorderLayout.java, 416–417
Borders, sharing, 424
BounceBallApp.java, 627
Bouncing ball (case study), 624–627
BallControl.java, 526–527
Ball.java, 625–626
BounceBallApp.java, 627
Boxing, 481
Braces, 16
break keyword, 135–139
  GuessNumberUsingBreak.java, 137
  TestBreak.java, 135–136
break statement, 94
BufferedInputStream, 659–660
BufferedOutputStream, 659–660
Bugs, 55
Bus, 2
Buttons, 572–578
  AbstractButton class, 514–515
  alignments, 574
  defined, 572
  icons, 572–573
  pressed icons, 572–573
  rollover icons, 572–573
  TestButtonIcons.java, 573–574
  text positions, 575
  using, 575–578
byte type, 32–33
Byte wrapper class, 477
Bytecode, 7
Bytecode verifier, 19
Bytes, 3

C

C#, 6
C, 6
Cable modem, 5
Calculator (problem), 321–322
  Calculator.java, 321–322
Calendar class, 462–463, 465
  field constants in, 463
  get(int field) method, 462
  TestCalendar.java, 463–465
Call stacks, 160
Calling methods, 158–160
Calling object, 271, 346, 380
capacity() method, StringBuilder, 318
Case sensitivity:
  of Java source programs, 14
  of name identifiers, 29
case statement, 94
Casting, *See* Type casting
Casting objects, 387–389
  CastingDemo.java, 388–389
catch block, 654
CD-R/CD-RW, 4
CDs/DVDs, 4
Central processing unit (CPU), 2–3
  sharing, 8
  speed, 3
Chained exceptions, 447–448
char (data type), 44–47
  casting between numeric types and, 46–47
  increment/decrement, 45
Character class, 313–315
  charValue method, 313
  CountEachLetter.java, 314–315
  methods for manipulating a character, 314
  static methods, 314
Character encoding, 45
Character literal, 45
Character wrapper class, 313–315
Characters:
  converting to strings, 309–310
  finding in strings, 308–309
charAt method, String class, 315–320
charAt(index) method, StringBuilder, 315–320
Check boxes, 578–581
  CheckBoxDemo.java, 579–581
  toggle button, 578
Checked exceptions, 439, 442
CheckPalindrome.java, 310–311
Child classes, *See* Subclasses
Choice lists, *See* Combo boxes
Circle class, 265–266, 270, 278–279, 281, 284–285, 289, 291
Circle1.java, 267–268
Circle2.java, 279
Circle3.java, 284–285
Circle4.java, 376–378
Circle.java, 460
Clarity, 415
  in class design, 363
Class abstraction, 307
Class block, 16
Class design guidelines, 362–364
  clarity, 363
  cohesion, 362
  completeness, 363
  consistency, 362
  encapsulation, 363
  inheritance vs. composition, 394
  instance vs. static, 363–364
  interfaces vs. abstract classes, 473–476
Class diagrams, 615
Class encapsulation, 347–351
class keyword, 19
Class loaders, 15
Class variables, *See* Static variables
ClassCastException, 387, 468
.class file, 15
Classes, 263–289
  Date class, 274–275
  Java, 265, 272, 276
  from Java library, 274–278
  naming, 52
  naming convention, 477
  Random class, 275
  reusing, 379
ClassNotFoundException, 437–438
Class's contract, 347
ClockAnimation.java, 559–560
clone method, Object class, 471–473
Cloneable interface, 471–473
Cloneable interface, House.java, 472–473
CloneNotSupportedException, 472–473
close() method, 326, 655
Close stream, 655, 657–658, 661
Closest pair, finding (problem), 242–244
  FindNearestPoints.java, 243–244
COBOL, 6
Code incrementally, use of term, 118
Code, modularizing, 165–167
Code sharing, and methods, 160
codebase attribute, 616
Cohesion, 362
  in class design, 362
Color class, 419
columns property, GridLayout manager, 413–415
Combining strings, 305–306

Combo boxes, 590–593
ComboBoxDemo.java, 591–593
defined, 590
JComboBox class, 590–591
Command-line arguments, 320–322
Comment styles, 51–52
Comments, 11, 51–52
Common design error, 281, 364
Communication devices, 2, 5
Comparable interface, 467–469
ComparableRectangle.java, 468–469
compareTo method, 304, 313
String class, 302–313
compareToIgnoreCase method, String class, 303–304
Comparing characters, 72
Comparison operators, 72
Compiler, 6–7, 17, 21
Compiling vs. interpreting, 7
Component class, 347
setFont method, 419, 422
Component classes, 406–407
ComponentAdapter class, 551
ComponentEvent, 534–535
ComponentListener interface, 537, 551
Components, naming convention for, 572
Composition, 353–355
ComputeArea class, 24, 29, 32
ComputeArea.java, 27–28
ComputeBMI.java, 84–85, 351
ComputeChange.java, 48– 50
ComputeFactorial.java, 679–681
ComputeFibonacci.java, 681–683
ComputeLoan.java, 38–39, 150, 172, 347
ComputeLoanUsingInputDialog.java, 56–57
Computer programming, defined, 2, 21
Computers:
central processing unit (CPU), 2–3
communication devices, 2, 5
defined, 2
input/output devices, 2, 4–5
memory (main memory), 2, 3
storage devices, 2, 4
ComputeTax.java, 86–88
Concatenating strings, 26, 50
Conditional expressions, 95
Conditional operator, 90, 95
Confirmation dialog, controlling a loop with, 139–140
Confirmation dialogs, 98–100
Consistency, in class design, 362
Console input, using Scanner class, 26–27
Console output, formatting, 95–97
Constants,
benefits of, 32
naming, 32
Constructor chaining, 380–382
Constructors, 270
compared to methods, 270
constructing objects using, 270
default, 270
no-arg, 270
overloaded, 270
Container class, 407, 410, 419
ContainerEvent, 534–535, 537
ContainerListener interface, 537, 551
Content pane delegation, 410
continue keyword, 135–137
TestContinue.java, 136–137
Contract, class, 347
Control unit, 2–3
Control variable, 117, 122, 126–128, 130, 220, 222
ControlBall.java, 626–627
Convenience adapters, 551
Convenience listener adapter, 551
Converting strings, 56
Copy reference, 208
Copying files (problem), 660
Copy.java, 661–662
CountLettersInArray.java, 213–215
Course class, 313–315
Course.java, 356–357
TestCourse.java, 356
Current time, displaying (case study), 37–39
currentTimeMillis method, 43
Custom exception classes, 448–450
Custom stack class, 393–394

D

Data encapsulation, benefit of, 362
Data field encapsulation, 283–286
Circle3.java, 284–285
TestCircle3.java, 285–286
Data fields, 265, 396, 466
hidden, 346
Data structures, 26
Data types, 29
DataInput interface, 655, 666
DataInputStream class, 655–657
TestDataStream.java, 657–658
using, 657
DataOutput interface, 655, 666
DataOutputStream class, 655–657
TestDataStream.java, 657–658
using, 657
Date class, 274–275
Date constructor, 274
De Morgan's law, 90
Debuggers, 55
Debugging, 54–55
defined, 55
Decimal numbers, 1319
conversions between binary numbers and, 1320
conversions between hexadecimal numbers and, 10–11
Declarations, 28, 29–30
Declared type, 385, 387–388
Declaring exceptions, 439–440
Decrement operator, 37–38
Deep copy, 473
Default constructor, 270
Default field values, 270
Default modifiers, 282
Default values, 200, 272–273, 288, 422, 593
Delete key, 5
Delimiters, changing, 307
Delphi, 6
Derived classes, *See* Subclasses
Descent, 510
Descriptive names, 25, 29, 52
full, 52
identifiers, 29
Deserialization, 664
destroy method, Applet class, 620
Dialup modem, 5
Directory path, 323
Directory size, finding (problem), 687–688
directory size, defined, 687
DirectorySize.java, 687–688
Disks, 4
DisplayImagePlayAudio.java, 634
DisplayImageWithURL.java, 633
DisplayMessage applet, parameter names/values for, 621
DisplayMessageApp.java, 623–624
DisplayMessage.html, 621–622
DisplayMessage.java, 622
DisplayTime.java, 34
DisplayUnicode.java, 41
Divide-and-conquer strategy, 177
Division (/), 33–34
Division by zero, and runtime errors, 54
do-while loop, 116–121
Documentation, 51–53
defined, 51
Dot operator (.), 271
Dot pitch, 5
Double precision, 25, 29, 33, 131, 478
double type, 32, 36
double type values vs. float type values, 35
Double wrapper class, 477
Double.parseDouble method, 56–57
draw3DRect method, , 502
drawArc method, 506
drawImage method, 521
drawLine method, 500–501
drawOval method, 506
drawPolygon method, 508–509
drawPolyline method, 508–509
drawRect method, 500
drawRoundRect method, 501
drawString method, 500
Drives, 4
Drop-down lists, *See* Combo boxes
DSL (digital subscriber line), 5
Duplicate objects, and object stream, 665
DVDs, 4
Dynamic binding, 385–387

E

Eclipse, 10, 14
Edible interface, 465, 476
Eight Queens puzzle, 230, 678
defined, 678
EightQueens.java, 695–697
isValid(row, column) method, 696
search(row) method, 696
Elementary programming, 25–66
Encapsulating string, 305
Encapsulation, 177, 274, 284–286, 305, 347–351
encoding, 38
Encoding scheme, 38–39
End-of-line style, 52–53
End tag, 616
enum keyword, 709
equalsIgnoreCase method, String class, 303
Erroneous solutions, 133

Error class, 437–438
  subclasses of, 438
Escape sequences for special characters, 46
Euclidean algorithm, 133
Event, defined, 484
Event-driven programming, 534–560
  defined, 534
EventObject class, 534
Events:
  key, 555–557
  mouse, 552–554
Exception class, 434, 438, 440–442, 448–450
Exception classes, types of, 437–439
Exception, defined, 435
Exception handling, 432–450
  advantages of, 434–437
  custom exception classes, 448–450
  exception types, 437–439
  finally clause, 445–446
  overview, 432–434
  Quotient.java, 432
  QuotientWithException.java, 433–434
  QuotientWithIf.java, 432–433
  QuotientWithMethod.cpp, 435–436
  rethrowing exceptions, 447
  when to use exceptions, 447
Exception propagation, 441, 447
Exception types, 437–439
Exceptions, 432–450
  chained, 447–448
  chained exceptions, 447–448
  checked, 439, 442
  declaring, 444–445
  propagation, 441, 447
  throwing, 444–445
  unchecked, 439, 445
Exponent methods, Math class, 172–175
Expression statement, 40
Expressions, 30
  conditional, 95
Extended classes, *See* Subclasses
Extending, prevention of, 396

F

factorial method, 679–681, 698
Factorials, computing:
  base case, 678
  ComputeFactorial.java, 679–681
  stopping condition, 678
Factorials, computing (problem), 678–681
FahrenheitToCelsius.java, 37
Fall-through behavior, 94
Fibonacci, Leonardo, 681
Fibonacci numbers, computing (problem), 681–683
  ComputeFibonacci.java, 681–683
  fib method, 683
FigurePanel class (case study):
  FigurePanel.java, 503–505
  TestFigurePanel.java, 502–503
  UML diagram, 265
File class, 322–325
  lastModified() method, 324
  obtaining file and directory properties using, 323
  TestFileClass.java, 324–325
File dialogs, 329–330
  ReadFileUsingJFileChooser.java, 329–330
File input and output, 325–329
  PrintWriter class, writing data using, 325–326
  Scanner class, reading data using, 326–327
  WriteData.java, 325–326
File names:
  absolute, 323
  relative, 323
File pointer, 666–668
FileInputStream class, 652–654
  constructors, 652
FileOutputStream class, 652–654
  constructors, 652
Files, naming consistently, 423
fill3DRect method, 502
fillOval method, 502
fillRect method, 501
fillRoundRect method, 502
Filter streams, 655
FilterInputStream class, 655
FilterOutputStream class, 655
final, 31
final modifier, 396
finally clause, 445–446
FindNearestPoints.java, 243–244
Flash drives, 4
float type, 27, 33
Float wrapper class, 477
Floating-point approximation, 34
Floating-point literals, 35
Floating-point numbers, 26, 33
  defined, 36
Floppy disks, 4
FlowLayout manager, 411–413, 417, 505
  placing components in a frame using, 410
  properties, 417
  ShowFlowLayout.java, 411–413
flush() method, 660
focusable property, 557
FocusEvent, 534–535, 537
FocusListener interface, 551
Font class, 419–420
  finding available fonts, 420
FontMetrics class, 510–512
  TestCenterMessage.java, 511–512
for-each loops, 203
for loop, 126–128
Formal parameters, 157
Format specifier, 95–97
FORTRAN, 6
Fractals:
  defined, 692
  problem, 692–695
  Sierpinski triangle, 692–695
    defined, 692
    SierpinskiTriangle.java, 692–695
Frames, 408–410
  adding components to, 410
  creating, 408–409, *See also* JFrame class
  MyFrame.java, 408–409
  MyFrameWithComponents.java, 410
Framework-based programming
  using, 536
Full descriptive names, 52
Function keys, 4–5
Functions, 157

G

Garbage collection, 208, 274
Garbage, defined, 274
Generic programming, 388–389, 480
  and interfaces, 480
GenericSort.java, 479–481
Geometric objects, 375
GeometricObject class, 375
  Circle4.java, 376–377
GeometricObject1.java, 375–376
  Rectangle1.java, 378
GeometricObject.java, 458–460
  abstract methods, 459
    benefits of defining, 460
  TestGeometricObject.java, 461–462
getArea() method, 264, 267, 272
  Circle class, 375
getArea() method, Rectangle class, 375
getAvailableFontFamilyNames() method, 420
getBMI() instance method, 352–353
getClass() method, 632
getDiammeter() method, Circle class, 375
getFontMetrics methods, Graphics class., 510
getHeight() method, 512, 520
get(int field) method, Calendar class, 462
getKeyCode() method, 555
getLocalGraphicsEnvironment() method, 420
getNumberOfObjects(), 278–279
getPerimeter() method:
  Circle class, 375
  Rectangle class, 375
getPreferredSize() method, 505
getRadius() method, 285
getSize() method, MyStack.java, 394
getSource() method, 535
getStatus() instance method, 353
getTime() method, Date class, 274
getWidth() method, 512, 520
GIF (Graphics Interchange Format), 422
Gigahertz (GHz), 3
Gosling, James, 8
goto keyword, 709
goto statement, 137
GradeExam.java, 241–242
Graphical user interface (GUI), 572
  See also GUI programming
  displaying components, 276
  GUIComponents.java, 277–278
  TestFrame.java, 276
Graphics, 498–525
  drawing on panels, 499–501
  graphical coordinate systems, 498
  Graphics class, 498–500
  images, displaying, 520–522
  ImageViewer class, 522–525
  MessagePanel class (case study), 512–516
  paintComponent method, 516
  StillClock class, 517–520

Graphics class, 498–500
  methods, 499, 508–509
GraphicsEnvironment class, 420
Greatest common divisor, finding (problem), 131–133
GreatestCommonDivisor.java, 132–133
GreatestCommonDivisorMethod.java, 165–166
GregorianCalendar class, 462–463
GridLayout manager, 413–415, 505
  properties, 417
  ShowGridLayout.java, 414–415
  specifying the number of rows/columns in a grid, 414
GuessBirthday.java, 74–78
GuessBirthdayUsingArray.java, 98–100
GuessBirthdayUsingConfirmationDialog.java, 250–251
GuessDate class:
  designing, 359–362
  GuessDate.java, 361–362
  UseGuessDateClass.java, 360–361
Guessing numbers (problem), 105–106
GuessNumber.java, 119–120
GuessNumberOneTime.java, 118–119
GuessNumberUsingBreak.java, 137
GUI programming, 406–424
  arcs, drawing, 506–507
  Color class, 419
  component classes, 406–407
  container classes, 406–407
  Font class, 408, 419–420,
  frames, 408–410
  GUI classes, classification of, 407
  helper classes, 406, 408
  image icons, 422–424
  Java GUI API, 406–408
  layout managers, 411–417
    BorderLayout manager, 415–417
    FlowLayout manager, 411-413
    GridLayout manager, 413–415
    properties of, 417
    setLayout method, 411, 420
  panels, using as subcontainers, 417–419
  polygons/polylines, drawing, 507–510
  Swing GUI components, 42
    common features of, 420–422
  Swing vs. AWT, 406

H

Hand-traces, 55
HandleEvent.java, 470–471
Handlers, 536
Handling the exception, use of term, 434
Hard disks, 4
Hardware, 27
Heavyweight Swing components, 406
Height, 510
Helper classes, 406, 408
Hertz (Hz), 3
Hexadecimal literals, 35
Hexadecimal numbers, 719
  conversions between binary numbers and, 719
  conversions between decimal numbers and, 719
hgap property:
  BorderLayout manager, 417
  FlowLayout manager, 417
  GridLayout manager, 417
Hidden data fields, 346
Hidden static methods, 383
High-level languages, 6–7
Histogram.java, 604–606
Horizontal alignment, 574, 584
Horizontal text position, 575
House.java, 472–473
howToEat method, 466, 475–476
hspace attribute, 617
HTML tag, 616
Hypertext Markup Language (HTML), 8

I

I/O, in Java, 650
Icons:
  alignments, 574
  sharing, 424
  text positions, 575
Identifiers, 29
  using abbreviations for, 52
IEEE 754, 33
if statements, 74–75
  nested, 80–81
  simple, 75
if...else statements, 79
IllegalArgumentException, 438, 440, 445
Image icons, 422–424
  TestImageIcon.java, 423–424
ImageObserver interface, 521
Images, displaying, 520–522
  DisplayImage.java, 521–522
ImageViewer class, 522–525
  ImageViewer.java, 523–525
  SixFlags.java, 523
Immutable classes, 344, 487
Immutable objects, 344–345
Immutable strings, 302–303
Implicit import, 17
Incompatible operands, 90
Increment and decrement operators, 40, 45
Incremental development and testing, 88, 183
Increment.java, 163
Indent code, 52
Indentation style, 52
Indexed variables, 200
indexOf method, String class, 308–309
Infinite loop, 117, 128
instanceof page 387
Infinite recursion, 679
Information hiding, 177
Inheritance, 374
  casting objects, 387–389
  defined, 374
  and modeling of the *is-a*relationship, 379, 394
  multiple, 379
  Object class, 384
  overriding methods, 382–383
  overriding vs. overloading methods, 383–384
  single, 379
  subclasses, 374–379
  super keyword, 380–382
  superclass, 374–379
init method, Applet class, 620
initial-action, 126–127
Inner class, 541–542
Inner class listeners, 541
InnerClass class, 541-542
Input dialogs, 55–58
  using, 56–57
Input errors, 54
Input/output devices, 2, 4–5
  keyboard, 4–5
  monitor, 5
  mouse, 5
Input stream, defined, 652
InputMismatchException, 327
InputStream class, 652
Insert key, 5
Insertion sort, 219–222
  InsertionSort.java, 222
Inside access, 283
Instance, 264
  static vs., in class design, 363
Instance method, 271
Instance variables, 278, 350
Instantiation, 264
int, 32
Integer literals, 35
Integer vs. decimal division, 37
Integer wrapper class, 476
Integer.parseInt method, 56
Integrated Development Environment (IDE), 10
  debugging in, 55
Intelligent guesses, 118
Interface inheritance, 465
Interfaces, 465–476
  abstract classes vs., 473–476
  ActionListener interface, 469–471
  Cloneable interface, 471–473
  defined, 465
  distinguishing from a class, 465
  and generic programming, 469
  marker, 471
  naming convention, 475
  TestEdible.java, 465–466
Interned strings, 302–303
Invoking a second constructor, 347
IOException class, 654
is-a relationships, 394, 475
is-kind-of relationship, 475
isDigit method, Character class, 314
isEmpty() method, MyStack.java, 394
isFocusable property, 557
isLetter method, Character class, 314
isLetterOrDigit method, Character class, 314
isLowerCase method, Character class, 314
isPalindrome method, 684
ItemEvent, 537
ItemListener interface, 537
Iteration of a loop, 116
Iteration, recursion vs., 697

J

JApplet class, 618
Java, 6
  anatomy of characteristics of, 8
  defined, 2, 12
  versatility of, 10
Java bytecode, running, 14

Java class, 265, 272
java ClassName, 15
java command, 15, 22
Java compiler, 12, 14
Java Development Toolkit (JDK), 10
Java Enterprise Edition (Java EE), 10
Java expressions, evaluating, 36–37
.java extension, 14
.java file, 14
Java GUI API, 406–408
Java keywords, *See* Keywords
Java language specification, 10–11
Java Micro Edition (Java ME), 10
Java modifiers, *See* Modifiers
Java program-development process, 13
Java program, simple, 11–13
Java source-code file, creating, 14
Java source programs, case sensitivity of, 14
Java Standard Edition (Java SE), 10
Java Virtual Machine (JVM), 14, 208
`java.applet`, 614
`java.awt` package, 406, 408
`java.awt.Color` class, 408, 419
`java.awt.Container`, 407, 410
`java.awt.Dimension`, 408
`java.awt.Font`, 408, 419
`java.awt.FontMetrics`, 408
`java.awt.Graphics`, 408, 499, 521
`java.awt.GraphicsEnvironment`, 420
`java.awt.LayoutManager`, 408
javac command, 15
javadoc comments, 51–52
`java.io`, 323
`java.io.FileNotFoundException`, 653
`java.io.IOException`, 653–654
`java.lang` package, 17, 39, 56, 313, 467
`java.lang.Class` class, 632
`java.lang.ClassNotFoundException`, 664
`java.lang.Comparable`, 467, 475
`java.math`, 481
`java.net`, 632
JavaServer Pages, 10
java.sun.com, 8–10, 51, 175
`java.util` package, 28
`javax.swing` package, 406,
`javax.swing.event` package, 596
`javax.swing.JApplet`, 407, 615
`javax.swing.JDialog`, 407
`javax.swing.JFrame`, 407
`javax.swing.JPanel`, 407
`javax.swing.ListSelectionModel`, 593
`javax.swing.Timer`, 557
`JButton` class, 276, 406, 410, 420, 422, 522, 535–536, 572–573, 575, 583
  GUIComponents.java, 277–278
`JCheckBox` class, 535, 572, 581
  GUIComponents.java, 277–278
`JComboBox` class, 276–277, 406, 535, 572, 590
`JComponent` class, 499, 501, 583–584
`JDialog` class, 406–407
JDK 1.6, 10
`JFileChooser` class, 330
  ReadFileUsingJFileChooser.java, 329–330
`JFrame` class, 276, 406–408, 410, 412–413, 420, 422, 547, 551, 587, 602, 615
  compared to `JApplet` class, 618
  `setDefaultCloseOpeartion` method, 276
  `setLocation` method, 276
  `setSize` method, 276
  `setTitle` method, 276
  `setVisible` method, 276
`JLabel` class, 406, 420, 422, 500, 583–584
`JList` class, 276, 535, 593–594
`JOptionPane` class, 50, 55
  `showMessageDialog` method, 29, 98
`JPanel` class, 277, 406–408, 417, 420, 500, 503, 521, 554
  as a canvas for drawing, 500
`JPasswordField` class, 572, 586
JPEG (Joint Photographic Experts Group), 422
`JRadioButton` class, 276, 406, 535, 578, 581
  GUIComponents.java, 277–278
`JScrollBar` class, 596
`JScrollPane` object, 586
`JSlider` class, 599
`JTextArea` class, 572, 576
  constructors/methods, 586
`JTextField` class, 406, 418, 420, 535, 572, 584–585
  GUIComponents.java, 277–278
`JToggleButton` class, 578

K

Key codes, 555
Key constants, 555
Key events, 555–557
  defined, 555
  KeyEventDemo.java, 556–557
`KeyAdapter` class, 551, 557
Keyboard, 4–5
`KeyEvent` class, 535, 537, 555
KeyEventDemo.java, 556–557
`KeyListener` interface, 537, 555
`keyPressed` handler, 555
`keyReleased` handler, 555
`keyTyped` handler, 555
Keywords, 11, 709

L

Labels, 583–584
  defined, 583
LAN (local area network), 5
LargeFactorial.java, 482
`lastIndexOf` method, `String` class, 308
`lastModified()` method, `File` class, 324
Layout managers, 411–417
  `BorderLayout` manager, 415–417
  `FlowLayout` manager, 411–413
    placing components in a frame using, 411
    ShowFlowLayout.java, 411–413
  `GridLayout` manager, 413–415
  `setLayout` method, 411, 420
Leading, 574–575
Leap year, determining (problem), 90–91
LeapYear.java, 91
Left associative operators, 97
Left justify, 97
`length()` method, `StringBuilder`, 318
Lightweight Swing components, 406–407
Line comment, 11
Line numbers, 11
Line separator string, 328
LinearSearch.java, 216–217
Lines, drawing, 501–502
`LinkageError` class, 438
Linux, 7
`ListenerClass` class, 470–471
Listeners, 469, 535–541
  anonymous class listeners, 542–544, 547
  defined, 535
  inner class listeners, 542, 544
  listener interface adapters, 536–537
  AdapterDemo.java, 551–552
Lists, 593–596
  defined, 593
  `JList` class, 593–594
  ListDemo.java, 594–596
`ListSelectionEvent`, 535
Literals, defined, 35
`Loan` class, 347–349, 467
  class diagram, 348
  Loan.java, 349–351
  test program, writing, 348–351
  TestLoanClass.java, 348–349
Loan payments, computing (case study), 37–39
Local variables, 153, 305
Logic errors, 54
Logical operators, 88–90
`long`, 32
`long` literal, 35
Long string, breaking, 26
`Long` wrapper class, 476
Loop body, 123
loop-continuation-condition, 116–117
Loop statement, 41
Loops, 103–139
  body, 123
  case studies, 131–135
  controlling, with a confirmation dialog, 139–140
  controlling with a sentinel value, 122–124
  defined, 116
  `do-while` loop, 124–126
  infinite, 117
  iteration of, 117
  `for` loop, 126–128
  nested, 129–130
  pre-test, 128
  `while` loop, 116–124
Loss of precision, 41, 49
Lottery (problem), 91–93
Lottery.java, 91–93

M

Mac OS, 7
Machine language, 5, 7, 15
Main class, 265
`main` method, 11–12, 15–16, 157–158, 266, 614, 618–621, 624, 634, 637
  passing strings to, 320–321
Main window, 602
Marker interface, 471
`matches` method, `String` class, 308
Matching braces, 12
`Math` class, 172–175, 272, 279, 396

abs method, 174
exp methods, 173
max method, 174
min method, 174
pow method, 172
random method, 174–175
rounding methods, 173–174
trigonometric methods, 172
Math learning tool, 73
advanced, 121–124
improved, 82–84
max method, 157–159, 468–469
Math class, 174
Mbps (million bits per second), 5
Megabyte (MB), 3
Megahertz (mHz), 3
Memory chips, 3
Memory (main memory), 3
Message dialog box, displaying text in, 16–17
MessagePanel class (case study), 512–516
implementation of, 513
MessagePanel.java, 513–516
TestMessagePanel.java, 513
xCoordinate property, 512
yCoordinate property, 512
Meta object, 633
Method abstraction, 176–183
bottom-up design, 179–180
PrintCalendar.java, 181–183
top-down design, 177–179
Method block, 12
Method header, 157
Method name, 157
Method overloading, 169
Method signature, 157
Methods, 16, 156–183, 271–272
body, 157
call stacks, 160
calling, 158–160
defined, 156–158
modularizing code, 165–167
naming, 52
overloading, 168–170
parameters, 157
passing arrays to, 209–212, 240–241
passing objects to, 286–287
passing parameters by values, 162–165
syntax for defining, 156
value-returning, 157
void, 160–162
Microsoft Windows, 6–7
min method, Math class, 174
Modem, 5
Modifier key, 5
Modifiers, 395–396, 714–715
methods, 157, 159
Modularizing code, 165–167
GreatestCommonDivisorMethod.java, 165–166
PrimeNumberMethod.java, 166–167
Monetary units, counting (case study), 47–50
Monitor, 5
Mouse, 5
Mouse events, 552–554
MoveMessageDemo.java, 553–554
moving a message on a panel using a mouse (example), 553–554
MouseAdapter class, 551
mouseDragged method, 554
MouseEvent, 535, 537
MouseListener interface, 552–553
MouseMotionAdapter class, 551, 554
MouseMotionListener interface, 536–537, 552–554
MovableMessagePanel class, 554
MoveMessageDemo.java, 553–554
Multidimensional arrays, 236–251
birth dates, guessing (problem), 250–251
Multimedia, 634–637
Multimedia animations, 634–637
Multiple-choice test, grading (problem), 241–242
GradeExam.java, 241–242
Multiple inheritance, 379
Multiple solutions, 133
Multiple windows:
creating, 602–606
Histogram.java, 604–606
MultipleWindowsDemo.java, 603–604
Multiplication (*), 33
MultiplicationTable.java, 129–130
Multiplicity, 354
Multiprocessing, 8
Multiprogramming, 8
Multithreading, 8
Mutator method, 284–285, 522
MyFrame.java, 408–409
MyFrameWithComponents.java, 410
MyStack.java, 393–394

N

Naming conflicts, avoiding, 266
Naming conventions, 475
Narrowing a type, 41
native keyword, 472
native modifier, 714
Nested blocks, 171
Nested class, 541–542
Nested if statements, 74, 80–81
Nested loops, 129–130
NetBeans, 10–11, 409
Network interface card (NIC), 2, 5
new operator, 199–201, 267, 270–271, 288
next(), 27
Next-line style, 52–53
nextByte() method, 27, 327
nextDouble() method, 27, 327
nextFloat() method, 27, 327
nextInt() method, 27, 327
nextLine() method, 27, 327–328
nextLong() method, 27, 327
nextShort() method, 27, 327
NIC, 5
No-arg constructor, 266
NoClassDefFoundError, 15
Nonextensible *is-a* relationship, 379
Nonserializable fields, 664
Nonstrict mode, floating-point arithmetic, 709
NoSuchMethodError, 15
NotSerializableException, 664
null keyword, 17
null value, 272–273
NullPointerException, 273, 438–439, 516
Number class, 477–478, 482, 486
Number systems, 717–≠720
Numbers, converting to strings, 56
Numbers, formatting, 42
Numeric data types, 32–37
Numeric errors:
avoiding, 131
minimizing, 130–131
Numeric keypad, 5
Numeric literals, 35–36
Numeric operators, 33–35
on characters, 47
Numeric types:
conversions, 41–42
range of, 41
Numeric values, converting to strings, 309–310
Numeric wrapper classes, 477–478
abstract methods implemented by, 478

O

Oak language, 8
Object class, 384,
clone method, 471
equals method, 384, 477
toString method, 384
Object I/O, 662–666
TestObjectInputStream.java, 663–664
TestObjectOutputStream.java, 663
Object member access operator (.), and casting operator, 345
Object-oriented design:
class design guidelines, 362–364
clarity, 363
cohesion, 362
completeness, 363
consistency, 362
encapsulation, 363
instance vs. static, 363
interfaces vs. abstract classes, 473–476
Rational class, 482–487
properties/constructors/methods, 483
Rational.java, 484–486
TestRationalClass.java, 483–484
Object-oriented programming:
class relationships, 353–355
aggregation, 353–355
association, 353–355
composition, 353–355
inheritance, 394
Object-oriented programming (OOP), 264, 276, 406
ComputeBMI.java, 84–85
defined, 264
Object serialization, 664
ObjectInputStream class, 662–663
Objects:
anonymous, 272
array of, 287–289
behavior of, 264
calling, 271
defined, 264
object reference variables vs., 271
passing to methods, 286–287
processing primitive data type values as, 476–479
state of, 270

Occurrences of letters, counting (case study), 212–215
CountLettersInArray.java, 213–215
Octal numbers, 35, 720
Off-by-one error, 203
Operands, converting, 41
Operating system (OS), 7–8, 14
defined, 7
scheduling operations, 8
system activities, controlling/monitoring, 7
system resources, allocating/assigning, 8
Operator associativity, 97–98
Operator precedence, 97–98
Operator precedence chart, 712–713
Option buttons, *See* Radio buttons
`OuterClass` class, 541
Output stream, defined, 650
`OutputStream` class, 653, 659
Ovals, drawing, 501–505
Overflow, 33
Overloaded constructors, 270
Overloading methods, 168–170
TestMethodOverloading.java, 169–170
Overriding methods, 382–383
Overriding, prevention of, 396

P

`pack()` method, 544, 586
Package-access, 282
Package-private, 282
Packages, using, 16, 282
Page Up/Page Dn keys, 5
`paintComponent` method, 499–500, 505, 516, 520–521, 523
invoking, 505
TestPaintComponent.java, 500–501
Palindromes, checking (problem), 310–311
Panels, using as subcontainers, 417–419
TestPanels.java, 417–419
Parameter list, 157, 169
Parameter order association, 162
Parameters, methods, 157
Parent classes, *See* Superclasses
`parseDouble` method, `Double` class, 478
`parseInt` method, `Integer` class, 478
Pascal, 6
Pass-by-sharing, 209, 287
Pass-by-value, use of term, 163
Passing arrays to methods, 209–212
TestPassArray.java, 211–212
Passing parameters by values, 162–165
Increment.java, 163
TestPassByValue.java, 163–165
Pixels, 5, 498, 500
Plus sign (+), 26
PNG (Portable Network Graphics), 422
`Point` class, 552, 694
Polygons/polylines, drawing, 507–510
DrawPolygon.java, 509–510
`polyLine` method, 528
Polymorphism, 384–385
defined, 384
dynamic binding, 385–387
generic programming, 388
matching vs. binding, 387
PolymorphismDemo.java, 385
`pop()` method, MyStack.java, 394
Post-test loop, 128
Postdecrement operator, 40
Postincrement operator, 40
`pow(a, b)` method, 37, 172
`printf` 95
Pre-test loop, 128
Precedence, 97
Predecrement operator, 40
Preincrement operator, 40
Prime numbers, displaying (problem), 137–139
PrimeNumber.java, 138–139
PrimeNumberMethod.java, 166–167
Primitive data types, 25
`print` method, 326
`println` method compared to, 28
PrintCalendar.java, 181–183
`PrintWriter` class, writing data using, 325, 650
`private` constructor, 283
Private constructor, 362
Private data fields, 344, 376, 379
`private` modifier, 282–283, 363, 395
Procedures, 157
Processing arrays, 201–203
Programming errors, 53–55
debugging, 54–55
logic errors, 54
runtime errors, 54
syntax errors, 53
Programming style, 51–53
defined, 51
Programs, 5–7,
creating/compiling/executing, 13–16
Properties, 264
Property default values, 422
`protected` modifier, 282, 396
Pseudocode, 24, 106
`public` modifier, 274, 282, 396
`push(Object)` method, MyStack.java, 393–394

Q

Quotient.java, 432
QuotientWithException.java, 433–434
QuotientWithIf.java, 432–433
QuotientWithMethod.java, 435

R

Radio buttons, 581–583
defined, 581
RadioButtonDemo.java, 582–583
radius data field, 264
Radix, decimal number system, 717
Ragged array, 238
RAM (random-access memory), 3
Random access files, 666–668
defined, 666
and processing of files of record, 668
TestRandomAccessFile.java, 667–668
`Random` class, 275
`random()` method, `Math` class, 82, 92
Random numbers, generating (case study), 175–176
RandomCharacter.java, 175–176
TestRandomCharacter.java, 176
`RandomAccessFile` class:
defined, 666
UML diagram, 666
`Rational` class, 483–487
limitations of, 487
overflow, 487
properties/constructors/methods, 483
Rational.java, 484–486
TestRationalClass.java, 483–484
ReadFileUsingJFileChooser.java, 329–330
`readObject()` method, 664
`Rectangle` class, 375, 377–378
Rectangle1.java, 378
TestCircleRectangle.java, 379
Rectangle.java, 460
Rectangles, drawing, 501–502
Recursion, 678–698
advantages of, 697
defined, 678
directory size, finding (problem), 687–688
Eight Queens puzzle, 695–697
defined, 695
EightQueens.java, 695–697
factorials, computing, 678–681
base case, 678
ComputeFactorial.java, 679–680
stopping condition, 678–679
Fibonacci numbers, computing (problem), 681–683
fractals (problem), 692–695
Sierpinski triangle, 692
SierpinskiTriangle.java, 692–695
infinite, 679
iteration vs., 697
overhead, 697
and performance concern, 697
problem solving using, 683–684
RecursivePalindromeUsingSubstring.java, 684
recursive helper methods, 684–687
binary search, 686–687
RecursivePalindrome.java, 684–685
selection sort, 685–686
Tower of Hanoi (problem), 688–691
Recursive call, 678
Recursive methods, 678
Recursive thinking, 683
RecursivePalindrome.java, 684–685
RecursivePalindromeUsingSubstring.java, 684
RecursiveSelectionSort.java, 685–686
Reference type, defined, 271
Reference variables:
accessing an object's data and methods, 271-272
accessing objects via, 270–274
Circle1.java, 267–268
defining classes and creating objects (example), 266
defined, 270
reference data fields and the null value, 272–273
and reference types, 273–274
TestCircle1.java, 266–267
variables of primitive types and reference types, differences between, 273–274
`regionMatches` method, `String` class, 304
Register listener, 536

Regular expression, 307
Relational operators, 72
Relative file name, 324
Remainder (%), 33–34, 712
repaint method, Component class, 505
replaceAll method, String class, 307–308
replaceFirst method, String class, 307–308
ReplaceText.java, 329
Replacing strings, 307
requestFocusInWindow() method, 586
Reserved words, 11
Resolution, 409
monitor, 5
Rethrowing exceptions, 447
return statement, 158–161
Return value type, methods, 157
Reusing methods, 160, 379
RGB model, 419
Right-associative operators, 98
Rounding methods, Math class, 173–174
rows property, GridLayout manager, 415
Runtime errors, 54

S

SalesTax.java, 42
Scanner class:
  console input using, 27
  how it works, 327–328
  input errors, avoiding, 328
  InputMismatchException, 327
  methods, 27, 326
  next() method/nextLine() method, 288
  nextByte() method, 27, 326
  nextDouble() method, 27, 326
  nextFloat() method, 27, 326
  nextInt() method, 27, 326
  nextLine() method, 27, 326
  nextLong() method, 27, 326
  nextShort() method, 27, 326
  ReadData.java, 327
  reading data using, 326–327
  ReplaceText.java, 328–329
  replacing text (problem), 289–290
Scientific notation, 36
Scope of variables, 171–172
Screen resolution, 5
Scroll bars, 596–599
  JScrollBar class, 596
  ScrollBarDemo.java, 598–599
Search word problem, 678
search(Object) method, MyStack.java, 393
Selection sort, 219–220, 685-686
  defined, 219, 685
  RecursiveSelectionSort.java, 685–686
  SelectionSort.java, 220
Selection statements, 72, 81–82
selectionMode, 593
Selections, 72–100
Semicolon (statement terminator), 11–12
Sentinel value, 122
SentinelValue.java, 123–124
SentinelValueUsingConfirmationDialog.java, 140
Sequential file, 666
Serializable interface, 664-665
  serializing arrays, 665–666
  TestObjectStreamForArray.java, 665–666
Serializable objects, defined, 664
Serialization, 664
setAlignment method, FlowLayout manager, 417
setBackground method, Component class, 419
setColor methods, Graphics class., 503
setColumns method, GridLayout manager, 417
setDefaultCloseOpeartion method, JFrame class, 276
setFont methods
  Component class, , 419, 516
Graphics class., 516,
setForeground method, Component class, 419
setHgap method:
  BorderLayout manager, 415
  FlowLayout manager, 411–413
  GridLayout manager, 413–415
setLayout method, 418–419
setlength method, StringBuilder, 317–318
setLocation method, JFrame class, 276
setLocationRelativeTo(null), 409
setRows method, GridLayout manager, 413–414
setSize method, JFrame class, 276, 409, 544
setSize(w,h) method, 409
Setter method, 284
setTitle method, 413
  JFrame class, 276, 408–409
setVgap method, BorderLayout manager, 417
setVgap method, FlowLayout manager, 417
setVgap method, GridLayout manager, 417
setVisible method, JFrame class, 276
Shallow copy, 473
Short-circuit operator, 90
short type, 27
Short wrapper class, 477
Shorthand operators, 39–41
ShowCurrentTime.java, 37–38
showInputDialog method, 55, 57
showMessageDialog method, JOptionPane class, 16
Sierpinski triangle, 692–694
  creation of, 692
  defined, 692
  SierpinskiTriangle.java, 692–694
    Point class, 694
Single inheritance, 379, 473
Single precision, 33
Single space, 52
Sliders, 599–602
  JSlider class, 599
  SliderDemo.java, 600–602
Software, 2, 5–7, 14
  defined, 5
Source file, 7, 14
Source object (source component), 469
Source program (source code), 6–7, 14
Spacing, 52–53
Special characters, escape sequences for, 46
Special floating-point values, 716
Specific import, 17
Specifier, 96–97
Speed, CPU, 3
Splash screen, 424
split method, String class, 307
Splitting strings, 307–308
sqrt method, 173
StackOfIntegers class, 316, 316–318
  StackOfIntegers.java, 359
  TestStackOfIntegers.java, 358–359
  UML diagram, 358
Stacks, 160, 357
start method, Applet class, 620
Start tag, 616
startAngle, 506
State, of objects, 264
Statement, 11
Statement terminator, 11
static inner class, 542
Static methods, 278
  hidden, 383
static modifier, 714
Static variables, 278–281, 345
  Circle2.java, 279
  TestCircle2.java, 279–281
Stepwise refinement, 177
StillClock class, 516–520, 559
  DisplayClock.java, 516–518
  StillClock.java, 518–520
  UML diagram, 517
stop method, Applet class, 620
Stopping condition, 678
Storage devices, 2, 4
Stream, defined, 650
Strict mode, floating-point arithmetic, 709
strictfp keyword, 709
strictfp modifier, 715
String class, 302–313
  characters, 302
  combining strings, 305–306
  compareTo method, 304
  as immutable class, 344
  methods for comparing strings, 303–304
  string length, 305–306
String concatenation operator, 26
String literal, 45
String type, 50–51
StringBuffer class, 315–320, 362
StringBuilder class, 315–320, 362
  capacity() method, 318
  charAt(index) method, 318
  constructors, 316
  initial capacity, 318
  length() method, 318
  methods for modifying string builders, 316–317
  setlength method, 317–318
Strings:
  calculator (problem), 321–322
  characters, finding in, 308–309
  CheckPalindrome.java, 310–311
  command-line arguments, 320–322
  comparisons, 304–305
  constructing, 302

conversion between arrays and, 309
converting, 307
converting characters and numeric values to, 309–310
converting to numbers, 56
immutable, 302–303
index range, 305
interned, 302–303
matching, by patterns, 307–308
PalindromeIgnoreNonAlphanumeric.java, 319–320
palindromes, checking (problem), 310–311
palindromes, ignoring nonalphanumeric characters when checking (problem), 318–320
passing to the main method, 320–321
replacing, 307-308
by patterns, 308
splitting, 307–308
by patterns, 308
string literal, 45, 305
string literal object, 302
string object, 302
string value, 302, 305
string variable, 302
substrings:
finding in, 308–309
obtaining, 306
Strong is-a relationship, 475
Stubs, 179–180
`Student` class, 272, 344, 386–387
Subinterface, 474
Subproblem, 139
`substring` method, `String` class, 306
Substrings:
finding in a string, 308–309
obtaining, 306
Subtraction (`-`), 33
SubtractionQuiz.java, 83–84
SubtractionQuizLoop.java, 121–122
Subtype, 384
Subwindows, 602
Sudoku game, developing an applet for, 614
Sudoku (problem), 244–248
CheckSudokuSolution.java, 245–247
Sun Java Website, 9–10
`super` keyword, 380–382
constructor chaining, 380–382
superclass constructors, 380
superclass methods, calling, 382
Supertype, 384–385
Supplementary Unicode, 45, 651
Swing:
AWT vs., 406
components, 406
Swing GUI components, 420–421, 572, 575
common features of, 420–421
TestSwingCommonFeatures.java, 421–422
`switch` statements, 74, 93–94
and `break` statement, 94
`synchronized` modifier, 715
Syntax errors, 53
Syntax rules, 12
System errors, 438
`System.currentTimeMillis()`, 37
`System.out.println`, 11–12

T

Tag, defined, 616
Tag name, defined, 616
Tapes, 4
computing (problem), 85–88
10BaseT, 5
Ternary operator, 95
TestBreak.java, 135–136
TestButtonIcons.java, 573–574
TestCenterMessage.java, 511–512
TestCircle1.java, 266–267
TestCircle2.java, 279–280
TestCircle3.java, 285–286
TestContinue.java, 136
TestCourse.java, 356
TestDoWhile.java, 125–126
TestEdible.java, 465–466
TestFileClass.java, 324–325
TestFrame.java, 276
TestGeometricObject.java, 461–462
TestImageIcon.java, 423
TestLoanClass.java, 348–349
TestMax.java, 158–160
TestMethodOverloading.java, 169–170
TestObjectInputStream.java, 663–664
TestObjectOutputStream.java, 663
TestObjectStreamForArray.java, 665–666
TestPaintComponent.java, 500–501
TestPanels.java, 417–419
TestPassArray.java, 211–212
TestPassByValue.java, 163–165
TestPassObject.java, 286–287
TestRandomAccessFile.java, 667–668
TestRandomCharacter.java, 176
TestStackOfIntegers.java, 358–359
TestSum.java, 130–131
TestSwingCommonFeatures.java, 421–422
TestVoidMethod.java, 160–162
TestWindowEvent.java, 549–551
Text areas, 586–589
DescriptionPanel.java, 587–589
`TextAreaDemo` class, 587
TextAreaDemo.java, 589
Text fields, 584–586
defined, 584
TextFieldDemo.java, 585–586
Text I/O, binary I/O vs., 650–652
Text I/O classes, 650
TextAreaDemo class, 587–588
Think before coding, use of term, 118
`this` keyword, 346
`this(arg-list)` statement, 347
`Throwable` class, 438, 442
Throwing exceptions, 440
`throws` Exception, 326–327
TicTacToe game (case study), 628–632
TicTacToe.java, 629–632
`Timer` class, animation using,557–560
AnimationDemo.java, 558–559
ClockAnimation.java, 559–560
`toCharArray` method, 309
Toggle button, 578
Token-reading methods, 327
`toLowerCase` method, `Character` class, 314–315
Tool tip, 420–421, 577
Top-down design, 177–179
`toString()` method:
`Date` class, 274–275
`GeometricObject` class, 382–383
MyStack.java, 393–395
TotalArea.java, 288–289
`toUpperCase` method, `Character` class, 307
Tower of Hanoi (problem), 688–691
defined, 688
TowersOfHanoi.java, 690-691
`transient` keyword, 664
`transient` modifier, 715
Trigonometric methods, `Math` class, 172
Truncation, 33, 41
`try-throw-catch` block, template for, 434
Two-dimensional arrays, 203–215
closest pair, finding (problem), 242–244
creating, 236–237
declaring variables of, 236–237
multiple-choice test, grading (problem), 241–242
obtaining the lengths of, 237
processing, 238–240
ragged array, 238
Sudoku (problem), 244–248
Type casting, 31, 410
defined, 41
explicit casting, 41–42
implicit casting, 47, 387
and loss of precision, 49
syntax for, 41

U

UML class diagram, 265
UML (Unified Modeling Language) notation, 265
Unary operator, 35
Unboxing, 481
Unchecked exceptions, 439
Underflow, 33
Unicode, 45
supplementary, 45, 651
Unix epoch, 43
URL class, 632–633
DisplayImagePlayAudio.java, 634
DisplayImageWithURL.java, 633
USB flash drives, 4
UseGuessDateClass.java, 360–361
User actions, source object and event type, 534
User interfaces:
buttons, 572–578
`AbstractButton` class, 572–573
alignments, 574
ButtonDemo.java, 576–578
defined, 572
icons, 572–574
pressed icons, 572–573
rollover icons, 572–573
TestButtonIcons.java, 573–574
text positions, 575
using, 575–578
check boxes, 578–581
combo boxes, 590–593
creating, 572–606
labels, 583–584
lists, 593–596

multiple windows, creating, 602–606
radio buttons, 581–583
scroll bars, 596–599
sliders, 599–602
text areas, 586–589
text fields, 584–586
UTF-8 format, 657

V

Value-returning method, 157–158, 317
valueOf method, String class, 309, 478
VarArgsDemo.java, 215–216
Variable-length argument lists, 215–216
Variables, 29–30
class's, 305
control, 127
declarations, 30
declaring/initializing in one step, 30
indexed, 198, 200
instance, 271, 278, 280–281, 346
local, 171, 272, 345, 396, 697
naming, 30, 52
scope of, 171, 345–346
static, 278–281, 345–346, 665
used as expressions, 31
Vertical alignment, 573–574, 584
Vertical text position, 575
vgap property:
BorderLayout manager, 415–416
FlowLayout manager, 411–413
GridLayout manager, 413–415
VirtualMachineError class, 437–438
Visibility, changing of, 396
Visibility modifiers, 282–283, 395
visible property, JFrame, 422
Visual Basic, 6
void keyword, 270
void method, 160–162
invoking, 161
return in, 161–162
TestVoidMethod.java, 160–162
vspace attribute, 617

W

Weak is-a relationship, 475
WelcomeInMessageDialogBox.java, 16–17
Welcome.java, 11–13
while loop, 116–124
Whitespace, 51
Whitespace character, 27, 51
Widening a type, 41
Wildcard import, 17
Window events, handling, 549–551
windowActivated method, 551
WindowAdapter class, 551
windowClosed method, 551
windowClosing method, 551
windowDeactivated method, 551
windowDeiconified method, 551
WindowEvent, 487, 551
windowIconified method, 551
WindowListener interface, 551
windowOpened method, 551
Windows Calculator, 718
World Wide Web (WWW), defined, 8
Wrapper class, 476–478
naming convention, 477
and no-arg constructors, 478
Wrapper class types, automatic conversion between primitive types and, 481
Wrapper objects, constructing, 478
Write-only streams, 666
writeBytes(String s) method, 656
writeChar(char c) method, 656
writeChars(String s) method, 656
WriteData.java, 325–326
writeUTF(String s) method, 656–656

X

xCoordinate, Component class, 512

Y

yCoordinate, Component class, 512

Z

Zero, division by, and runtime errors, 54
0 based array indices, 200